.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

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

13 de junio de 2024

Thiago Pacheco	Matias Bartellone	Iñaki Llorens
111003	110484	107552

1. Objetivos del trabajo

Los objetivos de este trabajo práctico son aplicar los conocimientos aprendidos en clase para analizar y demostrar que el problema planteado está en la clase NP, y demostrar también que posee la condición de ser NP-Completo. Así tambien como obtener la solucion óptima del problema NP-Completo, desarrollando algoritmos de Backtracking y Programación Lineal para luego compararlos. Por último se busca desarrollar algunos algoritmos de aproximación, estudiando su eficacia y sus ventajas y desventajas frente a los algoritmos que llevan a la solución real.

2. Análisis del problema

El problema planteado por la cátedra trata sobre cómo organizar la defensa de una ciudad en medio de una guerra. Como líderes estratégicos de la Tribu del Agua, debemos encontrar la solución a la estrategia propuesta por el maestro Pakku, que consiste en formar grupos de maestros agua de la manera más equitativa posible para que se turnen en las defensas y se mantenga una defensa constante.

Nuestra tarea consiste en determinar qué maestros formarán grupo con quién. Nuestras variables son:

- n: cantidad de maestros
- k: cantidad de sub-grupos que hay
- x_n: fuerza del maestro agua n
- S_k: sub-grupo de maestros k

Con los datos que se reciben al inicio del problema, nos corresponde analizarlos para encontrar un algoritmo que siempre encuentre la solución óptima. Es decir, debemos determinar cómo asignar a los maestros en cada subgrupo para que todos los subgrupos tengan una sumatoria equitativa, buscando que la diferencia de poder entre subgrupos sea mínima.

Para minimizar la diferencia entre grupos debemos tener en cuenta que:

- Los maestros pueden tener igual o diferente poder.
- Cada maestro debe y puede estar solamente en un sub-grupo
- Cada sub-grupo debe tener al menos un maestro.

Al analizar en profundidad el problema, se puede observar que existen varias formas de implementar la solución. En este trabajo práctico, implementaremos una solución óptima mediante backtracking, una aproximación utilizando programación lineal entera, otra utilizando un algoritmo greedy, y finalmente, la propuesta por el maestro Pakku. También demostraremos que el problema es NP-Completo y, por ende, presentaremos la correspondiente demostración de que es NP.

3. El Problema de la Tribu del Agua, es NP-Completo?

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

Para demostrar que el Problema de la Tribu del Agua se encuentra en NP, debemos mostrar que una posible solución puede ser verificada en tiempo polinomial. Entonces, tenemos que mostrar que dada una partición en k subgrupos $S_1, S_2, \ldots, S_k$, podemos verificar en tiempo polinomial si esta partición cumple con la condición de que la adición de los cuadrados de las sumas de las fuerzas de los grupos es menor o igual que B.

Para verificar esto entonces el proceso que se tiene que seguir es:

- Para cada subgrupo S_i, calcular la suma de las fuerzas de los maestros en ese subgrupo. Es $O(N)$ pq terminamos sumando todos los maestros.
- Elevar al cuadrado cada una de estas sumas. Es $O(K)$ pq hay K subgrupos.
- Sumar estos cuadrados. También es $O(K)$.
- Comparar el resultado con B para ver si cumple la condición. $O(1)$

Entonces la complejidad de la verificación es $O(N + K + K + 1)$, y nos queda $O(N + K)$. Si sucede que K es mucho menor que N nos quedaría $O(N)$.

Queda demostrado entonces que podemos verificar en tiempo polinomial si la propuesta de solución dada cumple o no, por lo tanto, el Problema de la Tribu del Agua se encuentra en NP.

3.2. 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 se debe reducir polinomialmente un problema X NP-Completo a este tal que $X \leq_p PR - TRIBU$. Entonces, se puede decir que el problema de la tribu es mayor o igual en dificultad que el problema reducido.

Proponemos utilizar el problema de las bolsas de supermercado, que es una variacion del problema de la mochila. Este problema en formato de decision plantea que dado una secuencia de n elementos $x_1, x_2, ..., x_n$, y dos numeros k y W definir si existe una particion en k bolsas tal que ninguna bolsa supere en valor al peso W.

Nuestro problema en formato de decision se plantea de esta manera: Dado una secuencia de n fuerzas/habilidades de maestros agua $x_1, x_2, ..., x_n$ y dos numero k y B, definir si existe una particion en k subgrupos $S_1, S_2, ..., S_k$ tal que:

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

Como puede observarse, los dos problemas son muy similares. Ambos tienen una secuencia de elementos, un numero que representa un valor limite y un numero k para determinar si se puede dividir en k subgrupos.

Buscamos usar la caja negra del problema de la tribu del agua para resolver el problema de las bolsas, por lo que hay que convertir los datos iniciales para poder usarse esa caja. Para la conversion de un problema a otro se deben realizar algunos cambios minimos en los datos de entrada, ya que tanto la secuencia de numeros como el numero k seran los mismos. Como en nuestro problema B representa la cota para la sumatoria de los cuadrados de k grupos y en el de las bolsas W representa la sumatoria de cada grupo, podemos definir $B = W^2 * k$

De esta manera, teniendo los datos transformados, se puede usar nuestro problema de la tribu del agua para resolver el problema de la bolsa llamandolo una sola vez. Podemos afirmar que se puede reducir polinomialmente este problema a nuestro problema original ya que lo unico que tuvimos que hacer fue modificar los datos de entrada, que en este caso cuesta $\mathcal{O}(1)$(un unico paso polinomial en la reduccion)

En conclusion, el problema de la tribu de agua es NP-Completo

4. Propuesta de Solución Backtracking

Realizamos un algoritmo de backtracking que obtiene la solución óptima al problema de la mínima sumatoria. Este algoritmo prueba todas las combinaciones posibles de los maestros agua en

K grupos, quedándose siempre con la solución de mínima suma. Para evitar analizar combinaciones que no conducen a una solución óptima, implementamos diversas condiciones de corte en nuestro programa. Dividimos nuestro algoritmo en dos secciones principales.

4.1. min sumatoria

Definimos las estructuras necesarias para almacenar la solución y otras variables relevantes para su uso en las llamadas recursivas. Inicializamos nuestro algoritmo con una solución greedy(explicada en el Item 6) como punto de partida para minimizar el tiempo de ejecución. Esta elección se debe a que una de nuestras estrategias de poda se basa en la mejor solución encontrada hasta el momento.

4.2. min suma

: es la funcion recursiva donde se buscaran todas las combinaciones posibles utilizando las podas para ahorrar tiempo. Tener presente que:

- **maestros** es la lista de todos los maestros
- **grupos** es una lista de listas de maestros, es decir una lista con cada sub-grupo.
- **n** es el índice con el que recorremos la lista de maestros.
- **sc** es una lista donde guardamos la solución(lista de sub-grupos de maestros), la sumatoria dada la solución actual y la cota superior.

Podas:

- **fuera del index** Si el índice excede la cantidad de maestros, realizamos un retroceso en la búsqueda.
- **grupos vacíos** Si nos encontramos en una situación donde tenemos menos maestros para colocar que grupos vacíos disponibles, retrocedemos en nuestra búsqueda. Es decir, si durante la generación de combinaciones nos quedan por colocar 2 maestros y aún hay 3 grupos vacíos, aplicamos un retroceso y volvemos a una etapa anterior de la búsqueda.
- **cota superior** Visualizamos cada sub-grupo de nuestra mejor solución actual en una recta según su sumatoria, lo que nos permite identificar un subgrupo con la suma mínima y otro con la suma máxima. Los demás subgrupos se encuentran distribuidos en el medio. LLamaremos cota superior al $S_(max)$

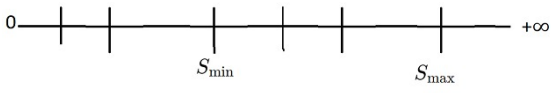

Figura 1: cota superior

  Durante la búsqueda de combinaciones de posibles grupos, si al agregar un maestro a un grupo se provoca que la suma del grupo supere la cota superior, resultando en una mayor diferencia entre los subgrupos y, por ende, en una solución peor a la que ya tenemos, entonces detenemos la búsqueda y retrocedemos.
- **supera sumatoria actual** Si la suma parcial actual es mayor que la suma de nuestra mejor solución encontrada hasta el momento (lo que indica que hemos encontrado una solución que será necesariamente peor), detenemos la búsqueda y retrocedemos.

- **ordenar grupos**: Ordenamos los grupos para que, al momento de asignar un maestro, siempre se agregue al grupo con la menor sumatoria hasta el momento. En caso de retroceder, se asignará al segundo grupo con la menor suma y se avanzará por esa rama. Aunque no se trate de una poda, esta estrategia tipo greedytacking aumenta la probabilidad de converger más rápidamente hacia la solución óptima.
- **recorte de combinaciones**: Al avanzar por el árbol de posibilidades, descartamos todas las ramificaciones que conducen a una solución idéntica a otra, pero con un orden diferente. Esto se debe a que la sumatoria de los subgrupos y la sumatoria final de estos son independientes del orden en el que se encuentren. Ejemplificamos:

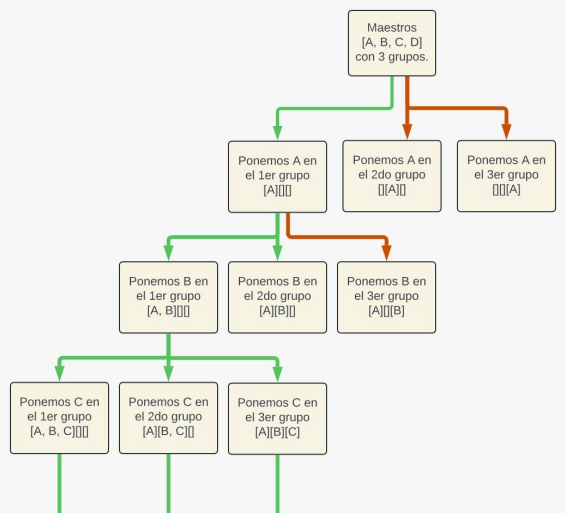

Figura 2: ejemplo arbol

Como podemos observar, al posicionar A en el primer grupo, obtenemos una configuración idéntica a la de posicionarlo en el segundo o en el tercer grupo. Por lo tanto, nos quedamos solo con una de esas opciones. Al posicionar B, notamos que la segunda rama es igual a la tercera, por lo que nos quedamos solo con una de esas dos.

```
1
2  def min_sumatoria(maestros, k):
3
4      grupos = [[] for _ in range(k)]
5
6      #los ordeno para que se recorten los primeroas ramas con valores altos
7      maestros = sorted(maestros, key=lambda x : x[1], reverse=True)
8
9      grupo_inicial = distribucion_inicial(maestros.copy(),k)
10
11     suma_total = sumatoria(grupo_inicial)
12
13     cota_sup = cota_superior(grupo_inicial)
14
15     sc = [grupo_inicial, suma_total, cota_sup]
16
17     _min_sum(maestros, grupos, 0, sc)
18
19     return sc[0], sc[1]
20
21 def _min_sum(maestros, grupos, n, sc):
22
23     if n > len(maestros):
24         return
25
```

```
    # si faltan colocar menos maestros que grupos vacios vuelve
    if grupos_vacios(grupos) > len(maestros) - n:
        return

    # si algun grupo supera la cota superior
    if superan_cota_superior(grupos,sc[2]):
        return

    # calculo la sumatoria parcial
    suma_parcial = sumatoria(grupos)

    # si mi sumatoria parcial es mayor vuelve
    if suma_parcial >= sc[1]:
        return

    # si ya use todos los maestros
    if n == len(maestros):
        # cambio si es mejor solucion
        if sumatoria(grupos) < sc[1]:
            sc[0] = [list(g) for g in grupos] #copia de array
            sc[1] = suma_parcial
            sc[2] = cota_superior(grupos) # actualizo la cota superior
        # si no vuelve
        return

    # Ordenar los grupos antes de a adir el siguiente maestro
    grupos.sort(key=lambda g: sum(y for (x, y) in g))

    # por cada grupo pruebo con el maestro actual o sin
    a = len(grupos)-1 - n if n <= len(grupos)-1 else 0 #recorte de posibilidades de
     primeros k grupos
    for g in grupos[a:]:
        g.append(maestros[n])
        _min_sum(maestros, grupos, n+1, sc)
        g.pop()

def grupos_vacios(grupos):
    return sum(1 for g in grupos if not g)

def sumatoria(grupos):
    return sum(pow(sum(y for (x, y) in g), 2) for g in grupos)

def superan_cota_superior(grupos, cota_sup):
    for g in grupos:
        if len(g) > 1 and sum(y for (x,y) in g) > cota_sup:
            return True
    return False

def cota_superior(grupos):
    grupo_max = max(grupos, key=lambda x : sum(y for (x, y) in x))
    return (sum(y for (x,y) in grupo_max))

def distribucion_inicial(maestros, k):
    grupos = [[] for _ in range(k)]
    maestros = sorted(maestros, key=lambda x: x[1], reverse=True)
    while (len(maestros) > 0):
        grupos.sort(key=lambda g: sum(y for (x, y) in g))
        grupos[0].append(maestros.pop(0))
    return grupos
```

4.3. Complejidad

La complejidad es $\mathcal{O}(n!)$. con n como la cantidad de maestros, ya que el algoritmo busca todas las posibles combinaciones de ubicación de los maestros en diferentes grupos, lo que podemos expresar como exponencial $\mathcal{O}(2^n)$. Aunque las podas mejoren el tiempo de ejecución, no alteran la complejidad.

5. Propuesta de Solución Programación Lineal

En programación lineal el método para resolver los problemas es diferente que cómo veniamos haciendo antes, esto se debe a que en lineal la dificultad del problema está en analizar nuestro problema original e identificar cuales son las variables de decisión y las restricciones, para luego pasarlas al resolvedor de PL. Para transformar el Problema de la Tribu del Agua para que pueda ser resuelto mediante programación lineal usamos la librería *pulp*, y como este es un problema de minimización vamos a definir al problema usando el atributo 'pulp . LpMinimize '. Dada la dificultad para volver lineal la función objetivo del Problema de la Tribu del Agua decidimos definir un modelo que resuelva una aproximación, esta es minimizar la diferencia del grupo de mayor suma con el de menor suma. Es decir, si el grupo Z es el de mayor suma de habilidades de los maestros ($\sum_i Z_i$) e Y es el de menor suma, entonces se busca minimizar $\sum_i Z_i - \sum_j Y_j$.

Necesitamos definir una serie de entidades (variables) que representen si un maestro i está o no en un grupo, vamos a definir entonces al inicio del problema las variables de decisión $x[i, j]$, estas son variables binarias que indicarán si el maestro i está asignado al grupo j. Se puede pensar similar a una matriz de adyacencia, donde hay un 1 si el maestro se encuentra y 0 si no se encuentra en el grupo j. Luego definimos las variables $suma_grupo[j]$, que son variables enteras que representan la suma de las habilidades de los maestros asignados al grupo j. Adicionalmente definimos las variables enteras Z e Y que representan la suma de habilidades del grupo con la mayor y menor suma, respectivamente. Luego pasamos a definir el problema que vamos a querer minimizar en nuestro modelo, y esto lo definimos haciendo que la función objetivo del resolvedor sea $Z - Y$. Teniendo definidas las variables y la función objetiva del modelo vamos a pasar a definir las restricciones y estas son:

- Cada maestro debe estar asignado a exactamente un grupo.
- Calcular la suma de habilidades para cada grupo.
- Definimos Z como la mayor suma de habilidades de los grupos.
- Definimos Y como la menor suma de habilidades de los grupos.

Por último, resolvemos el problema con el resolvedor CBC, que es el predeterminado, y parseamos la solución obtenida para devolver los maestros en los grupos como es esperado.

```python
def min_sumatoria_LP(maestros, k):
    n = len(maestros)

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

    prob = pulp.LpProblem("tribu del agua", pulp.LpMinimize)

    x = pulp.LpVariable.dicts("x", ((i, j) for i in range(n) for j in range(k)),
    cat='Binary') # 1 si el maestro i esta en el grupo j, 0 si no
    suma_grupo = pulp.LpVariable.dicts("suma_grupo", (i for i in range(k)),
    lowBound=0, cat='Integer') # suma de habilidades de los maestros en grupo j

    Z = pulp.LpVariable("Z", lowBound=0, cat='Integer')
    Y = pulp.LpVariable("Y", lowBound=0, cat='Integer')

    prob += Z - Y

    for i in range(n):
        prob += pulp.lpSum(x[i, j] for j in range(k)) == 1

    for j in range(k):
        prob += suma_grupo[j] == pulp.lpSum(maestros[i][1] * x[i, j] for i in range
    (n))

    for j in range(k):
        prob += Z >= suma_grupo[j]

```

```
    for j in range(k):
        prob += Y <= suma_grupo[j]

    solver = pulp.PULP_CBC_CMD(timeLimit=600)

    prob.solve(solver)

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

    print(f"Mayor suma (Z): {pulp.value(Z)}")
    print(f"Menor suma (Y): {pulp.value(Y)}")
    print(f"Diferencia (Z - Y): {pulp.value(Z) - pulp.value(Y)}")

    suma = sumatoria(grupos)

    return grupos, suma
```

Antes de pasar a las mediciones, cabe destacar que nuestro algoritmo presenta dificultades en su rendimiento cuando le toca resolver problemas con grandes cantidades de maestros. Esto se debe principalmente a que el algoritmo crea una variable por maestro por grupo, si hay N maestros y K grupos, se crean $N * K$ variables binarias, para luego probar todas las asignaciones posibles de las variables. Además, el algoritmo Simplex, cuando se adapta para manejar variables enteras funciona de la siguiente manera:

1. Primero resuelve el problema asumiendo que todas las variables son continuas, lo que proporciona una cota superior inicial para el valor objetivo.
2. Si alguna de las variables que deben ser enteras no resulta en un valor entero en la solución continua, el algoritmo entonces fuerza esa variable a ser entera y re-resuelve el problema.
3. Este proceso crea un árbol de decisiones, donde cada nodo representa una versión del problema con una variable entera fijada. El algoritmo explora este árbol en una búsqueda similar al backtracking.

Este árbol de decisiones va siendo exponencialmente más grande a medida que se agregan variables al problema, y la poda de ramas no siempre es eficiente. Es por esto que la complejidad algorítmica de resolver el Problema de la Tribu del Agua con programación lineal entera es exponencial, $O(2^n)$.

Para intentar resolver este problema probamos usando el solver GLPK, que tiene configuraciones específicas que optimizan la poda del árbol de decisiones, pero aún asi seguimos obteniendo los mismos resultados, el ejemplo dado por la cátedra '20-8.txt' llegó a tardar más de 24 horas sin llegar a la solución óptima. Es por esto que le agregamos al resolvedor de PL un límite de tiempo de 10 minutos, cuando llega a los 10 minutos de ejecución y todavía no ha encontrado la solución óptima, devuelve la solución más aproximada que tenga hasta el momento. Esta solución aproximada generalmente está muy cerca de la óptima que se obtiene en backtracking, que es la real. Luego, en la parte de mediciones, veremos todas las mediciones hechas pero cabe destacar que para el ejemplo '20-8.txt' la solución de backtracking da 11417428 y al de PL 11418372, lo cual está bastamte bien.

5.1. Mediciones de PLE y comparación con Backtracking

Usamos los ejemplos de prueba facilitados por la cátedra para medir y comparar los algoritmos de PLE y de Backtracking. Como explicamos anteriormente, y como se puede ver en la Figura 5, los tiempos de ejecución del algoritmo de lineal se vuelven exponenciales cuando el problema implica la creación de muchas variables es por esto que la mayoría de problemas llegan a los 600 segundos (10 minutos) de ejecución. Sin embargo la gran mayoría de problemas llegan a obtener la solución óptima al problema, esto se puede ver en las Figuras 3 y 4. Particularmente en la Figura

4 se muestran las diferencias entre las soluciónes obtenidas en Backtracking (que es el resultado óptimo) y las obtenidas por el algoritmo de lineal, como se puede ver en la mayoría de los casos la diferencia es 0, y en el peor de los casos no llega a una diferencia de 4000 en el coeficiente.

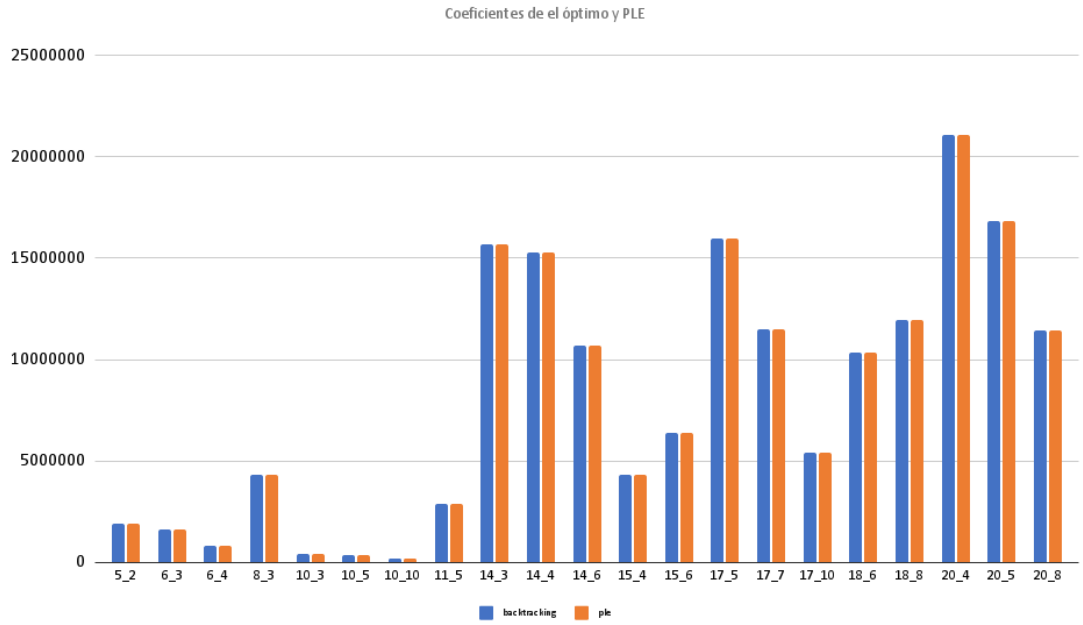

Figura 3: Coeficientes de el óptimo y PLE

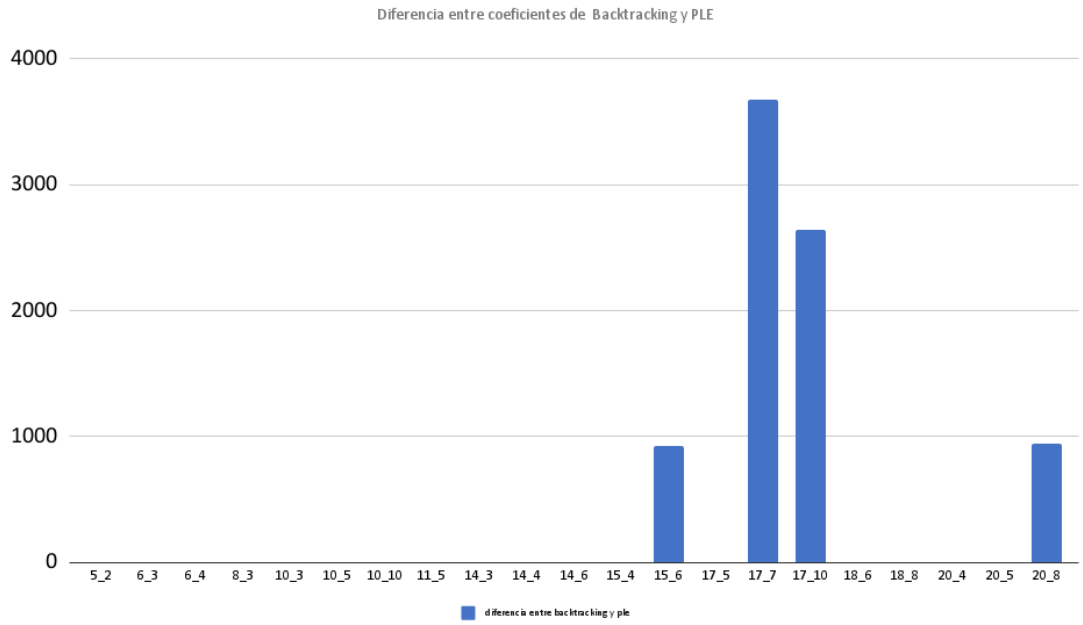

Figura 4: Diferencia entre coeficientes de Backtracking y PLE

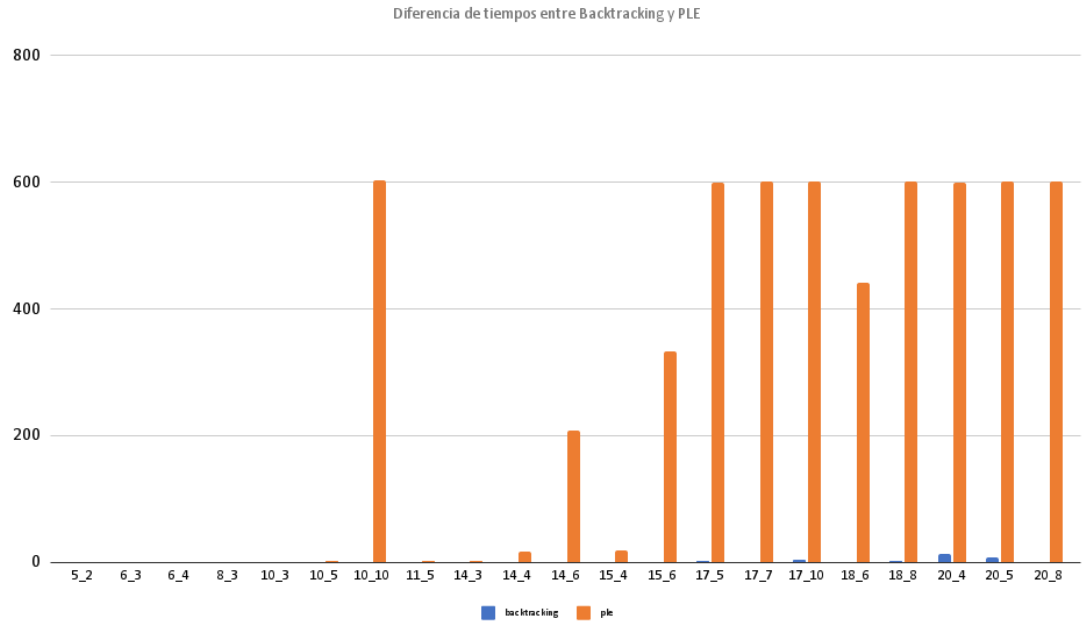

Figura 5: Diferencia de tiempos entre backtracking y PLE

6. Aproximación del Maestro Pakku

6.1. Algoritmo

A continuación presentamos la implementación del algoritmo:

```
def aproximacion_pakku(maestros, k):
    grupos = [[] for _ in range(k)]
    maestros = sorted(maestros, key=lambda x: x[1], reverse=True)
    while (len(maestros) > 0):
        grupos.sort(key=lambda g: sum(y for (x, y) in g))
        grupos[0].append(maestros.pop(0))
    return grupos
```

Básicamente, asignamos al maestro más poderoso sin grupo al grupo más débil, y repetimos este proceso hasta que no queden maestros por asignar.

6.2. Complejidad

La complejidad del algoritmo es $\mathcal{O}(nlog(n))$ porque debemos ordenar los maestros sumado a que por cada maestro buscamos el grupo de minima suma $\mathcal{O}(n.k)$. Por lo que nos queda $\mathcal{O}(nlog(n)) + \mathcal{O}(n.k)$. Donde k siempre sera menor o igual a n para que haya solución. Que lo podemos expresar como: $\mathcal{O}(n.log(n) + n.k)$

6.3. Análisis

A continuación presentamos la comparación entre la solución optima y la solución dada por la Aproximación de pakku en las pruebas dadas por la catedra.

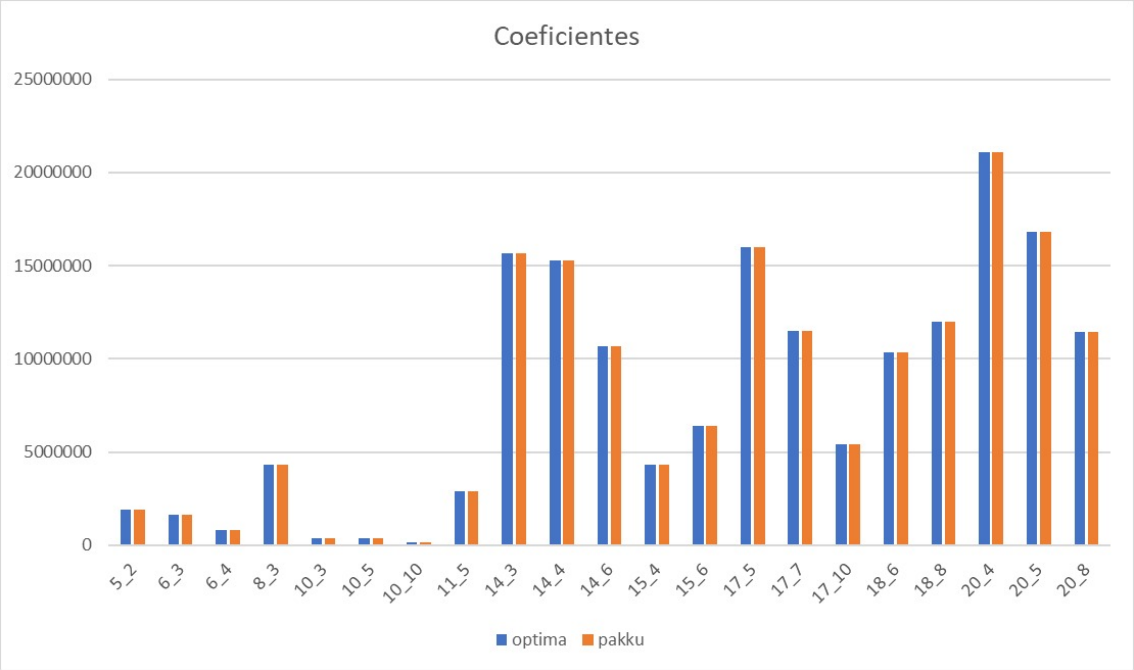

Figura 6: Coeficientes de solucion optima y por algoritmo de pakku

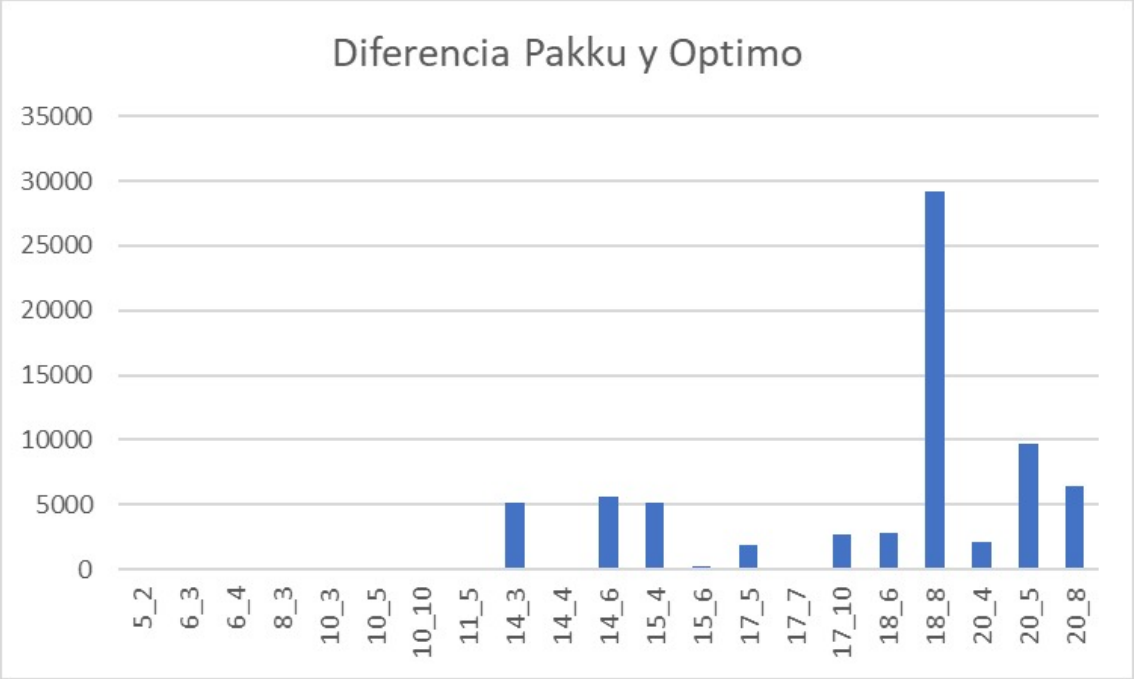

Figura 7: Diferencia de valores de Pakku respecto al optimo

Podemos observar que la diferencia varía según cada caso. Para mayor exactitud, mostramos la siguiente tabla, donde calculamos un Coeficiente r(A) para cada caso a través de A(I) sobre z(I), siendo z(I) el coeficiente de la solución óptima y A(I) el coeficiente de la solución del Algoritmo Pakku.

pruebas	optima	pakku	A(I) / z(I)
5_2	1894340	1894340	1
6_3	1640690	1640690	1
6_4	807418	807418	1
8_3	4298131	4298131	1
10_3	385249	385249	1
10_5	355882	355882	1
10_10	172295	172295	1
11_5	2906564	2906564	1
14_3	15659106	15664276	1,000330159
14_4	15292055	15292085	1,000001962
14_6	10694510	10700172	1,000529431
15_4	4311889	4317075	1,001202721
15_6	6377225	6377501	1,000043279
17_5	15974095	15975947	1,000115938
17_7	11513230	11513230	1
17_10	5427764	5430512	1,000506286
18_6	10322822	10325588	1,00026795
18_8	11971097	12000279	1,002437705
20_4	21081875	21083935	1,000097714
20_5	16828799	16838539	1,00057877
20_8	11417428	11423826	1,000560371
limite	79999200002	82221377780	1,0277775

Figura 8: Tabla de Calculos de r(A)

Como podemos ver al final, aparece la prueba límite. Esta es una prueba diseñada por nosotros donde, a través de un caso específico, tratamos de conseguir un error muy grande. La prueba consiste en buscar la solución para estos 5 magníficos maestros en 2 grupos.

MAESTROS	PODER
Iñaki	100.000
Felipe	99.999
Thiago	66.667
Matias	66.666
Buchwald	66.666

Figura 9: Datos prueba Limite

Donde cada solución sería:

	Solución Pakku					Solución Backtracking (óptima)		
GRUPO 1		GRUPO 2			GRUPO 1		GRUPO 2	
Iñaki	100.000	Felipe	99.999		Iñaki	100.000	Thiago	66.667
Matias	66.666	Thiago	66.667		Felipe	99.999	Matias	66.666
Buchwald	66.666						Buchwald	66.666
Diferencia entre grupos: 66.666					Diferencia entre grupos: 0			

Figura 10: Soluciones Prueba Limite

Y como podemos observar en la tabla(Figura 7), esta es la prueba donde obtenemos nuestro máximo valor de r(A), que utilizaremos como cota. Redondeando, nos quedaría que la cota es igual a 1.03.

Llegamos a esta distribución de la solución que encuentra la máxima diferencia entre la solución de backtracking y la de greedy, creando un algoritmo simple que se resuelve con programación lineal, este es:

```
import pulp

# Crear el problema de maximizaci n
problem = pulp.LpProblem("Maximize_e", pulp.LpMaximize)

# Definir las variables (enteras)
a = 100000  # a es una constante dada
b = pulp.LpVariable('b', lowBound=0, cat='Integer')
c = pulp.LpVariable('c', lowBound=0, cat='Integer')
d = pulp.LpVariable('d', lowBound=0, cat='Integer')
e = d  # e es igual a d

# Funci n objetivo
problem += e, "Maximize_e"

# Restricciones
problem += a + d == b + c, "Constraint_1"
problem += a + b == c + 2 * d, "Constraint_2"
problem += b >= c, "Constraint_3"
problem += c >= d, "Constraint_4"
problem += d >= 0, "Constraint_5"

problem.solve()

print(f"Estado: {pulp.LpStatus[problem.status]}")
print(f"b: {b.varValue}")
print(f"c: {c.varValue}")
print(f"d (e): {d.varValue}")
print(f"(a + d )** 2 - (b+c+e)**2: {(a + d.varValue) ** 2 - (b.varValue +c.varValue
    +e.varValue )**2}")
```

Creamos este algoritmo para el caso donde tenemos cinco maestros y queremos dividirlos en dos grupos, por eso las varibales representan los niveles de poder de cinco maestros y, inicializando el nivel del primer maestro a 100,000 encontramos que la máxima diferencia entre el algoritmo greedy y el de backtracking se obtiene con el segundo maestro teniendo 99,999, el tercero 66,667, el cuarto 66,666 y el quinto también 66,666.

Para corroborar también la cota calculada anteriormente realizamos un set de prueba con cantidades de datos inmanejables para el algoritmo de backtracking. Siguiendo los lineamientos que descubrimos anteriormente sobre cómo crear una prueba cuyo error sea máximo en el algoritmo greedy, creamos el set de prueba llamado 'grande-con-error-greedy.txt', este indica al algoritmo que cree dos grupos, y la cantidad de maestros es de 101, una cantidad muy grande de maestros que el algoritmo de backtracking tardaría mucho tiempo en calcular. Los primeros cuatro maestros tienen 1000, 900, 800 y 600 de poder, y luego todos los 97 restantes tienen 500. Por cómo funciona el algoritmo greedy el órden en que van a quedar los maestros es el primer grupo de la forma: $[1000, 600, 500...]$ y el segundo de la forma $[900, 800, 500...]$, por lo que el coeficiente resultante es 1341700000. Por supuesto que esta distribución no es la óptima, la distribución óptima sería de la forma [1000, 900, 500... (48 veces 500)] y [800, 600, 500... (49 veces 500)], ya que esta distribución hace que ambos grupos den la misma sumatoria de poder. Además esto daría un coeficiente de 1341620000.

Tenemos entonces $A(I) = 1341700000$ y $z(I) = 1341620000$, por lo tanto $\frac{A(I)}{z(I)} = 1{,}000059 \leq r(A)$. Por lo tanto corroboramos la cota calculada anteriormente con un ejemplo al que conocíamos la solución de antemano, pero que es imposible de llegar con el algoritmo de backtracking.

7. Algoritmo Greedy

El algoritmo greedy que implementamos consiste en ordenar a los maestros y asignar los primeros k maestros a un grupo cada uno. Luego, los últimos k maestros se distribuyen en los grupos, de manera que en el primer grupo se coloca el más grande con el más pequeño, en el segundo grupo el segundo más grande con el segundo más pequeño, y así sucesivamente. Este enfoque sigue un patrón de ida y vuelta. Ejemplificamos:

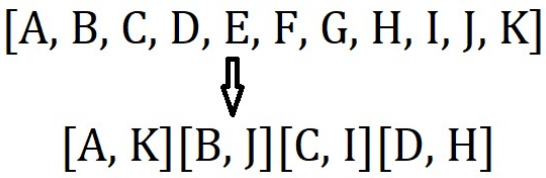

Figura 11: Primera Asignación de grupos

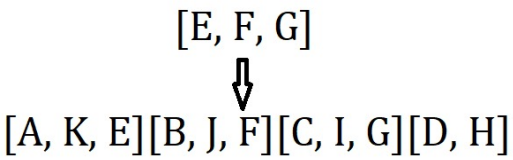

Figura 12: Segunda Asignación de grupos

7.1. Comparación con Aproximación del Maestro Pakku

A continuación, mostramos una primera tabla donde se presentan los coeficientes en cada prueba, incluyendo el coeficiente óptimo, el del Algoritmo Pakku y el de nuestro Algoritmo Greedy. Luego, presentamos una tabla donde se muestra la diferencia de cada método con la solución óptima.

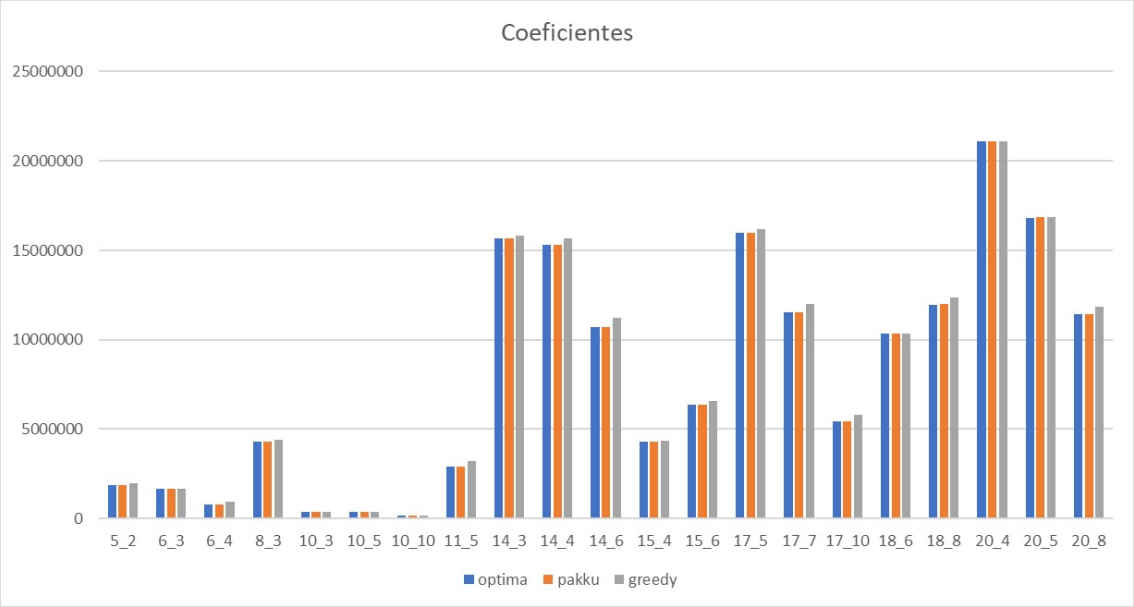

Figura 13: Coeficientes de la solución optima, algoritmo Pakku y nuestro algoritmo Greedy

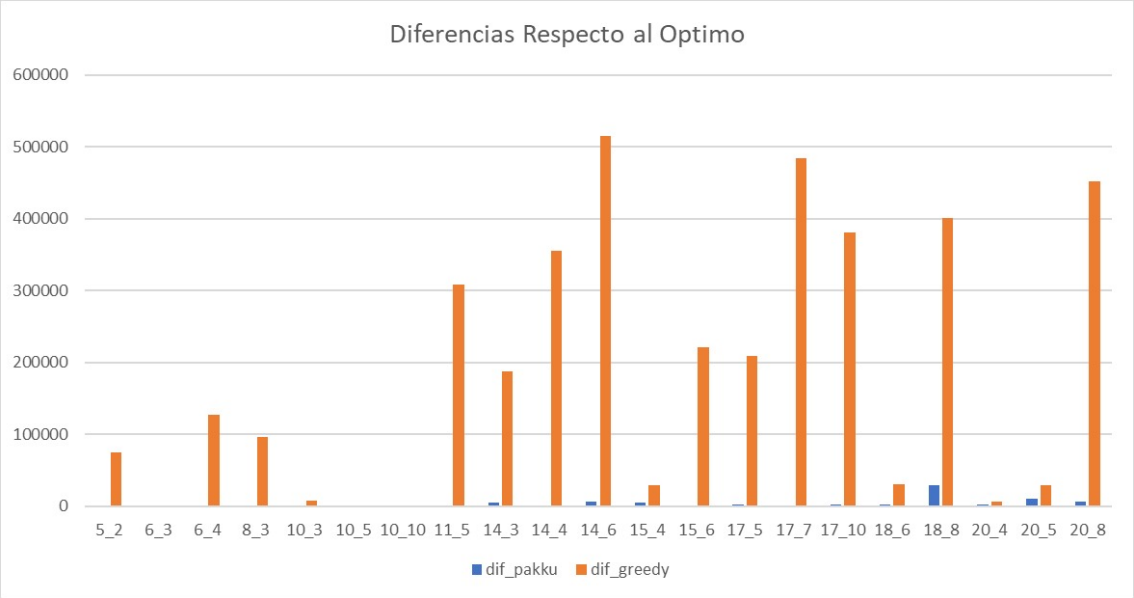

Figura 14: Diferencias con respecto al optimo del algoritmo Pakku y nuestro algoritmo Greedy

Como podemos observar, nuestra solución greedy parece ser una aproximación peor que la que nos proporciona el algoritmo del maestro Pakku.

7.2. Complejidad

Al ordenar los maestros según su fuerza, ya tenemos una complejidad de $\mathcal{O}(n \log(n))$. Luego, al recorrer la lista de maestros y asignarlos en grupos de a lo sumo k tandas(es decir agarramos k maestros y asignamos grupos), independientemente del orden, lo único que hacemos es asignar a

cada maestro un grupo, lo que resulta en una complejidad de $\mathcal{O}(n)$. Si recibimos los maestros ya ordenados, podríamos mantener esta complejidad.

8. Mediciones temporales

Se llevaron a cabo mediciones en segundos del rendimiento del algoritmo de backtracking utilizando las pruebas proporcionadas por la cátedra.

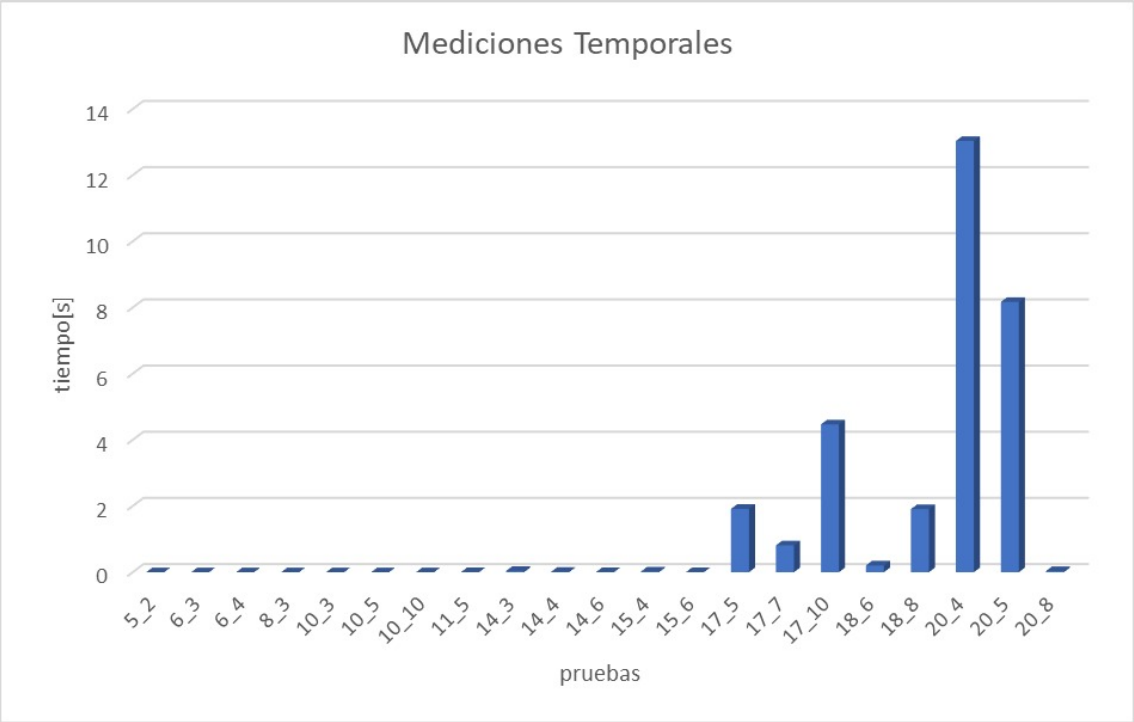

Figura 15: Mediciones temporales de las pruebas

Ploteamos los resultados de las mediciones en un gráfico para mostrar como varían los tiempos de ejecucion según la cantidad de elementos. Como se puede apreciar, el algoritmo no depende tanto de la cantidad de elementos en si, si no mas bien en la distribucion y valores de los elementos, ya que la prueba 20-4 tardo mas de 12 segundos cuando la de 20-8 no llega a 1. Esto se debe a que logro encontrar una solucion lo suficientemente optima al principio, lo que hace que se efectuen mas recortes en las ramas de recursion. Esto nos lleva a pensar que si los valores de poder de habilidad de los maestros se encuentran más dispersos entre sí, es decir que hay una cierta amplitud considerable entre los valores de poder de habilidad, es más facil para el algoritmo de backtracking encasillar a los maestros, porque al intentar equilibrar la suma de los poderes encuentra más facilmente las distribuciones óptimas de como poner a los de habilidad más baja con los de habilidad más alta. En cambio, si los valores de poder de los maestros no varían tanto se le dificulta al algoritmo encontrar las distribuciones óptimas.

9. Conclusiones

En conclusion este trabajo práctico explora la resolución del problema de la Tribu del Agua mediante distintos metodos. A través de la implementación del algoritmo de backtracking nos enfrentamos a un desafío sifnificativo en terminos de rendimiento, ya que es complejo manejar grandes

cantidades de datos debido a la complejidad exponencial del mismo. A pesar de las dificultades, los resultados obtenidos son optimistas respecto a nuestras espectativas previas. Ademas se muestran metodos como la aproximacion del Maestro Pakku y el algoritmo greedy, que ofrecen una solucion mas rapida pero menos precisa en ciertos casos.