FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TP3: Problemas NP-Completos para defender la Tribu Agua

[75.29] Teoría de Algoritmos
Primer Cuatrimestre de 2024

Integrantes

Alumno	Padrón
FUENTES, Azul	102184
GALIÁN, Tomás Ezequiel	104354
SANTANDER, Valentín	105637

Índice

1. Introducción

La Nación del Fuego está planeando un ataque a la Tribu Agua. Para que la tribu pueda defenderse de la manera más óptima posible, el maestro Pakku ha reclutado diferentes guerreros los cuales tienen asignado un nivel de fuerza/habilidad. En otras palabras, se cuenta con una lista de maestros (que denotaremos M) de tamaño $n \in \mathbf{N}$ donde cada elemento es una tupla con el nombre del maestro y su nivel de habilidad (x). Para el i-ésimo maestro, su nivel de habilidad será x_i. Los maestros que tengan el mismo nombre serán enumerados para poder diferenciarlos.

$$M = [(\text{Nombre\_1}, x_1), (\text{Nombre\_2}, x_2), \cdots, (\text{Nombre\_n}, x_n)]$$

La estrategia que ha decidido utilizar Pakku consiste en dividir a sus tropas en una cantidad fija de grupos donde la suma de la fuerza/habilidad de los maestros de cada grupo tiene un nivel lo mas similar posible.

Nuestra tarea es utilizar distintas estrategias de diseño de algoritmos para poder determinar el armado de los grupos de la manera más óptima posible. Sin embargo, como se verá en secciones posteriores, este problema no tiene una solución sencilla. En primer lugar se demostrará que la división de grupos que nos solicita el maestro Pakku es un problema NP Completo. Luego se mostrarán soluciones con mayor o menor optimalidad utilizando Backtracking, Programación Lineal y finalmente un enfoque heurístico o Greedy.

2. Demostración: El problema de la Tribu Agua es NP Completo

Se comenzará el análisis del problema realizando una demostración de que el problema de la Tribu Agua es NP Completo. Para ello se utilizarán dos problemas auxiliares. Estos son Multiway Number Partition Problem y Subset Sum Problem.

El objetivo final es demostrar que el problema de la Tribu Agua es NP Completo reduciéndolo polinomialmente al problema de Multiway Partition. Pero para que la demostración sea correcta se debería demostrar antes que Multiway Partition Problem es NP Completo pues no es un problema que se haya visto durante el curso.

Sin embargo el problema de Subset Sum es un problema NP Completo y si ha sido visto en clase. Por lo tanto se partirá de la base de que lo es sin necesidad de demotrarlo con una reducción polinomial y se utilizará para demostrar que Multiway Partition Problem es NP Completo.

De esa manera por transitividad se demostrará que el problema de la Tribu Agua es NP Completo
En otras palabras, denotando a los problemas como:

- Subset Sum Problem (SS)
- Multiway Number Partition Problem (MW)
- Problema de la Tribu Agua (TA)

Se realizarán las siguientes reducciones polinomiales:

$$\text{SS} \leq_p \text{MW}$$
$$\text{MW} \leq_p \text{TA}$$

2.1. Demostración: Multiway Partition Problem es NP Completo usando Subset Sum Problem

La reducción que se desea realizar es la siguiente:

$$\text{SS} \leq_p \text{MW}$$

Para demostrar que MW es NP Completo es necesario que las siguientes afirmaciones sean ciertas.

1. MW $\in$ NP
2. Las soluciones del problema de decisión de MW dan una solución para SS siempre que MW tenga solución.

Primero se definen ambos problemas en su versión de problema de decisión.

Subset Sum Problem (SS)

Dado un conjunto de números no negativos S y un valor objetivo t, ¿existe un subconjunto S' de S tal que la suma de los elementos de S' sea igual a t?

Multiway Number Partition Problem (MW)

Dado un conjunto de números enteros positivos A y un entero positivo k, ¿es posible dividir A en k subconjuntos disjuntos de manera que la suma de los elementos en cada subconjunto sea la misma?

Luego se analiza la veracidad de cada postulado para probar la demostración.

2.1.1. Multiway Partition Problem pertenece a NP

Un problema se encuentra en NP (Clase de Problemas No Deterministas Polinomiales) si se puede verificar una solución propuesta en tiempo polinomial.

```
1  def validar_multiway_partition_problem(A, k, subconjuntos):
2
3      def contar(conjunto):
4          contador = {}
5          for elemento in conjunto:
6              contador[elemento] = contador.get(elemento, 0) + 1
7          return contador
8
9      if len(subconjuntos) != k:
10         return False
11
12     elementos_cubiertos = []
13
14     suma_objetivo = sum(subconjuntos[0])
15
16     for subconjunto in subconjuntos:
17         suma = 0
18         for elemento in subconjunto:
19             if elemento not in A:
20                 return False
21             elementos_cubiertos.append(elemento)
22             suma += elemento
23
24         if suma != suma_objetivo:
25             return False
26
27     return contar(A) == contar(elementos_cubiertos)
```

En respuesta a eso se propone el algoritmo anterior implementado en Python que verifica si una solución de MW es correcta o no. Si puede decidirlo en tiempo polinomial, entonces MW pertenece a NP.

```
9      if len(subconjuntos) != k:
10         return False
11
12     elementos_cubiertos = []
```

Primero se verifica que la cantidad de subconjuntos proporcionada sea igual a la cantidad de partes a la que se quiere dividir el conjunto A. Si eso no coincide, no puede darse como válida la solución y por eso se devuelve `False`. En Python la cantidad de elementos de las estructuras de datos propias del lenguage se mantiene actualizada, por lo tanto esta operación se realiza en tiempo constante. Además se inicializa una lista de elementos cubiertos que se utilizará después para garantizar la cobertura total y disjunta de los elementos de cada subconjunto con respecto al conjunto A.

```
14     suma_objetivo = sum(subconjuntos[0])
```

Luego se suman todos los elementos del primer subconjunto de la lista de subconjuntos. De esa manera se obtiene el valor objetivo que deben tener todos los demás subconjuntos. Esta operación se realiza en tiempo lineal pues para sumar todos los elementos del subconjunto se debe iterar necesariamente cada uno de ellos para poder agregarlo en suma. El subconjunto más grande de cualquier conjunto es el conjunto universal, por lo tanto la suma no puede ser más compleja que $O(n)$ siendo n la cantidad total de elementos de A.

```
16      for subconjunto in subconjuntos:
17          suma = 0
18          for elemento in subconjunto:
19              if elemento not in A:
20                  return False
21              elementos_cubiertos.append(elemento)
22              suma += elemento
23
24          if suma != suma_objetivo:
25              return False
```

Ahora bien, para poder verificar que todos los subconjuntos suman lo mismo, se iteran los elementos de cada subconjunto y se calcula la suma total. Si el valor de esa suma es diferente al valor objetivo calculado antes, entonces no se cumplen las condiciones de MW y se devuelve `False`.

Mientras se calcula la suma, se verifica si alguno de los elementos del subconjunto no está en el conjunto A. En caso de que ocurra, la solución planteada no respeta las condiciones del problema y se devuelve `False`. Además se van agregando los elementos a la lista de elementos cubiertos.

Todas las operaciones para cada elemento de cada subconjunto son constantes (sumar una variable, agregar un elemento en una lista y consultas booleanas). Por lo tanto la complejidad estará marcada por la cantidad de veces que se itere cada elemento de cada subconjunto. En el peor de los casos, cada subconjunto podría ser el conjunto A completo pues es el subconjunto con mayor cantidad de elementos. Por lo tanto, se puede afirmar que la complejidad será $O(k' \cdot n)$ donde k' es la cantidad de subconjuntos de la solución propuesta y n es la cantidad total de elementos de A.

```
27      return contar(A) == contar(elementos_cubiertos)
```

Finalmente se realiza una comparación entre el conjunto total de elementos y los elementos cubiertos calculados anteriormente. Si ambos tienen los mismos elementos con la misma cantidad de apariciones, eso significa que la suma de todos los subconjuntos es igual a A y que cada elemento pertenece a un solo subconjunto.

La función `contar()` toma una lista de elementos y devuelve un diccionario donde las claves son los elementos y los valores la cantidad de apariciones que tienen en la lista. Comparara que dos diccionarios sean iguales tiene complejidad lineal pues se debe iterar cada elemento del diccionario y verificar si existe en el otro diccionario y sus claves son iguales. En caso de que ambos diccionarios sean iguales, se iterarán todos los elementos resultando en una complejidad $O(n)$ donde n es la cantidad de elementos de cualquiera de los diccionarios.

```
3       def contar(conjunto):
4           contador = {}
5           for elemento in conjunto:
6               contador[elemento] = contador.get(elemento, 0) + 1
7           return contador
```

A su vez, `contar()` construye cada diccionario de ocurrencias sumando las claves por cada aparición de cada elemento. Esto tiene también un comportamiento lineal análogo al análisis anterior ($O(n)$ donde n es la cantidad de elementos de la lista).

Cabe añadir que el conteo de ocurrencias siempre se realiza dos veces (Una para el conjunto A y otra para el conjunto de elementos cubiertos). Por lo tanto la complejidad de la linea final sigue siendo $O(n)$ pues se realiza un algoritmo de complejidad lineal sobre los elementos tres veces. Una para calcular las ocurrencias de A, otra para calcular las ocurrencias de los elementos cubiertos y una final para verificar que son iguales.

En conclusión, como las partes con mayor complejidad temporal del algoritmo tienen complejidad lineal, se puede deducir que el validador verifica la solución en tiempo lineal o bien polinomial. Esto demuestra que MS pertenece a NP.

2.1.2. Multiway Partition Problem soluciona el problema de decisión de Subset Sum Problem

Para terminar la demostración, es necesario probar que el problema de decisión de SS puede ser resuelto utilizando un algoritmo que resuelve el problema de decisión para MW. Para ello, se propone lo siguiente.

```
1 def problema_decision_subset_sum(S,t):
2
3     z_1 = sum(S)
4     z_2 = 2*t
5     S_prima = S + z_1 - z_2
6
7     return problema_decision_multiway_number_partition(S_prima,2)
```

Al conjunto de valores que se usará para el problema de decisión de MW se lo denotará como S'. Este conjunto es igual al conjunto S que provee el problema de decisión de SS pero con dos elementos más. Por un lado z_1 que es igual a la suma de todos los elementos de S y por otro lado z_2 que es el doble de t. Como indica el problema de decisión de SS, t es el valor objetivo que debe tener la suma de los elementos de al menos uno de los subconjuntos de S. Se pide resolver el problema de decisión de MW con $k = 2$. En otras palabras, se pide partir el conjunto S' en dos subconjuntos cuya suma sea igual.

Estas de decisiones aparentemente caprichosas, resuelven el problema de decisión de SS por la siguiente demostración.

Primero, conviene definir que n es la cantidad de elementos de S y que cada elemento de S se puede representar como s_i donde $1 \leq i \leq n, i \in N$. Se procede análogamente para S'. Además se definirá la suma de todos los elementos de S como z_1.

$$z_1 = \sum_{i=1}^{n} s_i$$

Luego, la suma de todos los elementos de S' es:

$$\sum_{i=1}^{n+2} s'_i = s_1 + s_2 + \cdots + s_n + z_1 + z_2 = \sum_{i=1}^{n} s_i + z_1 + z_2 = z_1 + z_1 + z_2 = 2z_1 + z_2 = 2z_1 + 2t$$

Por lo tanto, si el problema de decisión de MW es cierto, eso significaría que la suma de los elementos de cada subconjunto es $z_1 + t$ pues se debe dividir en dos subconjuntos con la misma suma.

Esta situación implicaría necesariamente que uno de los subconjuntos contiene a z_1 y el otro contiene a z_2 pues $z_1 + z_2 = z_1 + 2t$ lo cual haría imposible que se verifique el problema de decisión si ambos están en el mismo subconjunto pues en este caso se sabe que la suma total de los elementos es $z_1 + t$.

Teniendo en cuenta que la suma de los elementos de los dos subconjuntos del problema de decisión de MW es $z_1 + t$ y que uno contiene a z_1 y el otro a z_2 se concluye inmediatamente que se soluciona el problema de decisión de SS. Esto es así porque, suponiendo que se obtiene el subonjunto de S' que contiene a z_1, si retiramos este elemento, la suma de los elementos restantes es t que era el problema de decisión original. Por otro lado, si se retira z_2 del subconjunto restante, tendrá como suma total $z_1 - t$ que sería el complemento del conjunto solución que suma t. Como se vuelve a las condiciones del problema original con la solución correcta, se resuelve el problema de decisión también para SS.

En caso de que el problema de decisión de MW no verifique la solución, se sabe que para SS tampoco la tiene pues dadas las condiciones solo se verifica en caso de que exista también para MW.

Finalmente, queda demostrado que MW es un problema NP Completo pues se puede reducir polinomialmente al problema de SS que es sabido que es NP Completo.

2.2. Demostración: El Problema de la Tribu Agua es NP Completo usando Multiway Partition Problem

Ahora, la reducción que se desea realizar es la siguiente:

$$\text{MW} \leq_p \text{TA}$$

Para demostrar que TA es NP Completo es necesario que las siguientes afirmaciones sean ciertas.

1. TA $\in$ NP
2. Las soluciones del problema de decisión de TA dan una solución para MW siempre que TA tenga solución.

Primero se definen ambos problemas en su versión de problema de decisión.

Multiway Number Partition Problem (MW)

Dado un conjunto de números enteros positivos A y un entero positivo k, ¿es posible dividir A en k subconjuntos disjuntos de manera que la suma de los elementos en cada subconjunto sea la misma?

Problema de la Tribu Agua (TA)

Dada una secuencia de n fuerzas/habilidades de maestros agua $x_1, x_2, \cdots, x_n$ y dos numeros k y B, definir si existe una partición en k subgrupos $S_1, S_2, \cdots, S_k$ tal que:

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

Cada elemento x_i debe estar asignado a un grupo y solo a un grupo

Luego se analiza la veracidad de cada postulado para probar la demostración.

2.2.1. El Problema de la Tribu Agua pertenece a NP

```python
def validar_problema_de_la_tribu_agua(X,k,B,S):

    def contar(conjunto):
        contador = {}
        for elemento in conjunto:
            contador[elemento] = contador.get(elemento, 0) + 1
        return contador

    if len(S) != k:
        return False

    suma_total = 0
    fuerzas_cubiertas = []

    for s in S:
        suma_s = 0
        for fuerza in s:
            if fuerza not in X:
                return False
            fuerzas_cubiertas.append(fuerza)
            suma_s += fuerza
        suma_total += suma_s**2

    return contar(X) == contar(fuerzas_cubiertas) and suma_total <= B
```

En caso de que este algoritmo implementado en Python verifique si una solución de TA es correcta o no en tiempo polinomial, quedará demostrado que el problema pertenece a NP. Cabe aclarar que `X` es la secuencia de fuerzas de los maestros agua, k es la cantidad de subgrupos, B es el valor objetivo que debe tener la sumatoria y S es el conjunto con los subgrupos.

```python
    if len(S) != k:
        return False

    suma_total = 0
    fuerzas_cubiertas = []
```

Primero se verifica que la el largo del conjunto de subgrupos coincida con el valor k pasado por parámetro. De lo contrario no se cumplen las definiciones del problema y se debe devolver `False`. Además se inicializa el valor de la sumatoria total en 0 y se crea una lista donde se irán agregando los valores de cada subgrupo.

De manera análoga al MW, todas estas operaciones se realizan en tiempo constante.

```
15      for s in S:
16          suma_s = 0
17          for fuerza in s:
18              if fuerza not in X:
19                  return False
20              fuerzas_cubiertas.append(fuerza)
21              suma_s += fuerza
22          suma_total += suma_s**2
```

Luego se itera cada valor de fuerza de cada subconjunto. Si alguno de los valores de fuerza de los sugrupos no está en la secuencia de maestros, no se cumplen las definiciones del problema y se debe devolver `False`. Además se agrega cada valor de fuerza a la lista creada anteriormente y se suma la fuerza del maestro actual a la suma de fuerzas del subgrupo. Luego de que se haya iterado cada maestro del subgrupo, se suma el cuadrado de la suma del subgrupo a la sumatoria total.

También de manera análoga a MW este recorrido en caso de que se cumplan todas las condiciones, la complejidad temporal es $O(n)$ donde n es la cantidad de valores de fuerza en la secuencia X. Esto es así porque todos los maestros aparecen una sola vez en uno solo de los subconjuntos. Se puede entender como si fuera la misma lista partida en diferentes sublistas.

```
15      return contar(X) == contar(fuerzas_cubiertas) and suma_total <= B
```

Finalmente se verifica que la cantidad de apariciones de los elementos de las fuerzas cubiertas sea igual a la cantidad de apariciones de la secuencia original. Además de eso, la suma total debe ser menor al valor objetivo B.

Como ya se ha analizado en MW, esta operación se realiza en tiempo lineal. La complejidad en este caso sería $O(n)$ donde n es la cantidad de valores de fuerza en la secuencia X.

```
2       def contar(conjunto):
3           contador = {}
4           for elemento in conjunto:
5               contador[elemento] = contador.get(elemento, 0) + 1
6           return contador
```

Como ya se ha analizado en MW, el costo de crear un diccionario de ocurrencias en una lista de elementos es lineal.

Como todas las secciones del algoritmo tienen complejidad temporal lineal o constante (o bien polinomiales) el algoritmo completo tiene complejidad polinomial. Por lo tanto, como la verificación de una solución propuesta corre en tiempo polinomial, el problema de la Tribu Agua pertenece a NP. Esto tiene sentido pues es similar a Multiway Number Partition Problem en el objetivo de separar un conjunto de enteros en secciones disjuntas cuyos elementos suman cantidades iguales o similares.

2.2.2. El Problema de la Tribu Agua soluciona el problema de decisión de Multiway Partition Problem

Para terminar la demostración, ahora hay que probar que MW puede ser resuelto utilizando un algoritmo que resuelve el problema de decisión para TA. Para ello se propone lo siguiente:

```
1       def problema_decision_multiway_number_partition(A,k):
2
3       if sum(A) % k != 0:
4           return False
5
6       B = sum(A)**2//k
7
8       return problema_decision_tribu_agua(A,k,B)
```

En primer lugar, se debe revisar que la división entre la suma total de todos los elementos y la cantidad de subconjuntos que se desea obtener no tenga resto. De lo contrario no se cumpliría que cada subconjunto sume la misma cantidad.

En caso de que eso se cumpla, el valor objetivo que garantiza que MW es cierto cuando se cumple TA es $B = \frac{S^2}{k}$ donde S es la suma de todos los elementos de A. Esta decisión se justifica por lo siguiente.

Primero, se parte de la definición de la desigualdad de Cacuchy-Schwarz la cual se asume demostrada. Esta establece que para todo conjunto de números reales $a_1, a_2, \cdots, a_n$ y $b_1, b_2, \cdots, b_n$, se cumple que:

$$\left(\sum_{i=1}^{n} a_i b_i\right)^2 \leq \left(\sum_{i=1}^{n} a_i^2\right)\left(\sum_{i=1}^{n} b_i^2\right)$$

Teniendo en cuenta que S_i corresponde al valor de la suma de todos los elementos del i-ésimo subconjunto de A, se realizan los siguientes reemplazos:

- $a_i = 1 \quad \forall i : 1 \leq i \leq k$
- $b_i = S_i \quad \forall i : 1 \leq i \leq k$

Esto resulta en:

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

Denotando a la sumatoria del valor total de todos los subconjuntos como S:

$$S = \sum_{i=1}^{k} S_i = \sum_{i=1}^{k}\left(\sum_{e_j \in S_j} e_j\right)$$

Y despejando, se llega a la siguiente desigualdad

$$\sum_{i=1}^{k} S_i^2 \geq \frac{S^2}{k} \implies \sum_{i=1}^{k}\left(\sum_{e_j \in S_j} e_j\right)^2 \geq \frac{S^2}{k}$$

Teniendo esto en cuenta se puede inferir que el menor valor posible que puede tener la suma de los cuadrados del valor total de cada subconjunto es $\frac{S^2}{k}$.

Si se elige $B = \frac{S^2}{k}$ como valor objetivo en el problema de decisión de TA y devuelve `True`, eso garantizaría que:

$$\sum_{i=1}^{k}\left(\sum_{e_j \in S_j} e_j\right)^2 \leq B \implies \sum_{i=1}^{k}\left(\sum_{e_j \in S_j} e_j\right)^2 \leq \frac{S^2}{k}$$

Cabe aclarar que se pasan los mismos elementos del problema de decisión de MW por eso la desigualdad es cierta para los e_j definidos antes. Como la sumatoria está acotada inferior y superiormente por el mismo valor, la expresión se convierte en una igualdad.

$$\frac{S^2}{k} \leq \sum_{i=1}^{k}\left(\sum_{e_j \in S_j} e_j\right)^2 \leq \frac{S^2}{k} \implies \sum_{i=1}^{k}\left(\sum_{e_j \in S_j} e_j\right)^2 = \frac{S^2}{k}$$

En otras palabras, si el problema de decisón de TA con el valor objetivo $B = \frac{S^2}{k}$ es verdadero, entonces la suma de los cuadrados de la suma total de los subconjuntos de A es igual a $\frac{S^2}{k}$

Ahora bien, si los elementos de todos los subconjuntos de A sumaran la misma cantidad, su suma debería ser $\frac{S}{k}$ pues:

$$\sum_{i=1}^{k}\left(\sum_{e_j \in S_j} e_j\right) = \frac{S}{k} + \frac{S}{k} + \cdots + \frac{S}{k} = k \cdot \frac{S}{k} = S$$

Y la suma de cuadrados sería:

$$\sum_{i=1}^{k} \left(\sum_{e_j \in S_j} e_j \right)^2 = \left(\frac{S}{k}\right)^2 + \left(\frac{S}{k}\right)^2 + \cdots + \left(\frac{S}{k}\right)^2 = k \cdot \left(\frac{S}{k}\right)^2 = \frac{S^2}{k}$$

Queda entonces demostrado que con el valor objetivo $B = \frac{S^2}{k}$ se resuelve el problema de decisión de MW solo cuando se verifica correctamente el problema de decisión de TA. Por lo tanto, se puede reducir Multiway Number Partition (un problema NP Completo) al problema de la Tribu Agua. Finalmente, queda demostrado que el problema de la Tribu Agua es NP Completo.

3. Solución Propuesta: Backtracking

3.1. Implementación

El algoritmo utilizado para obtener las soluciones del problema de la Tribu Agua utilizando Backtracking es el siguiente:

```
1  def calcular_coeficiente(sumas):
2      return sum(suma**2 for suma in sumas)
3
4  def indices_ordenados_por_valor(arreglo):
5      tuplas_indice_valor = list(enumerate(arreglo))
6      tuplas_ordenadas = sorted(tuplas_indice_valor, key=lambda x: x[1])
7      indices_ordenados = [tupla[0] for tupla in tuplas_ordenadas]
8      return indices_ordenados
9
10 def buscar_optimo_tribu_agua_bt(maestros, habilidades, k, grupos,/
11                                 sumas, index, mejor_solucion):
12     if index == len(maestros):
13         coeficiente_actual = calcular_coeficiente(sumas)
14         if coeficiente_actual < mejor_solucion[0]:
15             mejor_solucion[0] = coeficiente_actual
16             mejor_solucion[1] = [list(grupo) for grupo in grupos]
17         return
18
19     for i in indices_ordenados_por_valor(sumas):
20         if index == 0 and i >= 1:
21             break
22
23         if index > 0 and any(sumas[i] == sumas[j] for j in range(i)):
24             continue
25
26         maestro = maestros[index]
27         habilidad = habilidades[index]
28         sumas[i] += habilidad
29
30         if calcular_coeficiente(sumas) < mejor_solucion[0]:
31             grupos[i].append(maestro)
32             buscar_optimo_tribu_agua_bt(maestros, habilidades, k,/
33                                         grupos, sumas, index + 1, mejor_solucion)
34             grupos[i].pop()
35
36         sumas[i] -= habilidad
37
38 def p_opt_tribu_agua_bt(maestros, habilidades, k):
39     pares_ordenados = sorted(zip(maestros, habilidades),/
40                              key=lambda x: x[1], reverse=True)
41     maestros, habilidades = zip(*pares_ordenados)
42     mejor_solucion = [float('inf'), None]
```

```
43      grupos = [[] for _ in range(k)]
44      sumas = [0] * k
45
46      buscar_optimo_tribu_agua_bt(maestros, habilidades, k, grupos,/
47                                  sumas, 0, mejor_solucion)
48      return mejor_solucion[1], mejor_solucion[0]
```

3.2. Explicación del algorítmo

Antes de explicar la implementación de nuestro algoritmo de backtracking, recordemos la "fórmula"general para resolver estos problemas. Esta consiste en los siguientes pasos:

1. Si se encuentra una solución, se devuelve (y se termina si es el caso).
2. Se avanza si es posible.
3. Se verifica si la solución parcial es válida:
   a) Si no es válida, se retrocede y se vuelve al paso 2.
   b) Si es válida, se llama recursivamente y se vuelve al paso 1.

En nuestro caso, buscaremos la mejor forma de distribuir n maestros en k grupos, de manera que se minimice la suma de los cuadrados de las sumas de las fuerzas de los grupos:

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

Para obtener este valor, podríamos probar todas las posibles distribuciones de los n maestros en los k subconjuntos y quedarnos con la mínima. Por ejemplo, si contamos con 3 maestros $[M1, M2, M3]$ con habilidades $[H1, H2, H3]$ para ubicar en 2 grupos, las combinaciones posibles serían:

$$S_1 = [M1], \quad S_2 = [M2, M3]$$

$$S_1 = [M2], \quad S_2 = [M1, M3]$$

$$S_1 = [M3], \quad S_2 = [M1, M2]$$

$$S_1 = [], \quad S_2 = [M1, M2, M3]$$

Es importante destacar que formar la opción $S_1 = []$ y $S_2 = [M1, M2, M3]$ es equivalente a $S_1 = [M1, M2, M3]$ y $S_2 = []$, ya que ambas opciones darán el mismo coeficiente. Esta observación será crucial más adelante.

Una vez analizadas todas las opciones, podemos comparar cuál de ellas tiene el menor coeficiente para así obtener el mínimo. Sin embargo, analizar el problema de esta forma sería más cercano a una lógica de fuerza bruta y no aprovecharía la estrategia de backtracking.

Transformemos esta idea en un algoritmo de backtracking. La idea es ubicar a los maestros en los grupos (ya veremos cómo), de manera que cada vez que se coloca un maestro se analiza si la solución parcial es válida. Una solución será válida si su coeficiente parcial, calculado para los grupos en su estado actual, es menor que el de la mejor solución encontrada hasta el momento. Es decir, si mientras armamos los grupos aún podemos llegar a una mejor solución que la actual, seguimos por ese camino. Si vemos que la solución parcial no puede superar la mejor solución encontrada previamente, retrocedemos un paso. Retroceder implica sacar el maestro del grupo en el que fue colocado y probarlo en otro grupo donde aún no haya sido ubicado. Si se han probado todas las opciones para un maestro, retrocedemos un paso más y reubicamos al maestro anterior, repitiendo el proceso hasta agotar todas las posibilidades. Si vemos nuevamente los pasos quedaría algo de la siguiente forma:

1. Si ya ubiqué a todos los maestros, evalúo la solución obtenida:

   a) Si el coeficiente es menor que el de la mejor solución encontrada anteriormente, la guardo.
   b) Si no, la descarto.

2. Agrego un maestro a un grupo.
3. Verifico si el coeficiente parcial no supera la mejor solución:
   a) Si el coeficiente parcial no es válido, quito el maestro y vuelvo al paso 2.
   b) Si es válido, llamo recursivamente y vuelvo al paso 1.

Miremos cómo sería con el ejemplo anterior. Para mayor claridad, asignamos valores a las habilidades, siendo estas $[1, 2, 3]$. Supongamos que colocamos a los maestros en el mismo orden descripto:

- Comienzo con el maestro $M1$ y lo ubico en el grupo $S1$: $S_1 = [M1], \quad S_2 = []$. El coeficiente sería 1. Como todavía no tengo una posible solución (o mi posible solución es muy grande), puedo seguir construyendo. Llamo recursivamente.
- Ahora tomo al maestro $M2$ y lo ubico también en el grupo $S1$: $S_1 = [M1, M2], \quad S_2 = []$. Mi coeficiente pasa a ser 9, y al igual que en el punto anterior, continúo.
- Tomo al maestro $M3$ y lo ubico nuevamente en el grupo $S1$: $S_1 = [M1, M2, M3], \quad S_2 = []$. Mi coeficiente pasa a ser 36, y al igual que en el punto anterior, continúo. Llamo recursivamente.
- Como ya no tengo más maestros que ubicar, calculo si la solución a la que llegué es menor que la mejor encontrada hasta el momento. Como es así, mi mejor solución pasa a ser 36 y retrocedo para explorar otras posibles soluciones.
- Retrocedo un paso, quitando al maestro $M3$ del grupo $S1$ y probándolo en el grupo $S2$: $S_1 = [M1, M2], \quad S_2 = [M3]$. El coeficiente parcial es 14. Como 14 es menor que 36, continúo. Llamo recursivamente.
- Como ya no hay más maestros que ubicar, evalúo esta solución. El coeficiente es 14, que es menor que 36, así que actualizo mi mejor solución a 14. Retrocedo de nuevo.
- Como ya no tengo otras posiblidades de ubicar a mi maestro $M3$ en esta configuración, retrocedo otro paso, quitando al maestro $M2$ del grupo $S1$ y probándolo en $S2$: $S_1 = [M1], \quad S_2 = [M2]$. El coeficiente parcial es 5. Como 5 es menor que 14, continúo. Llamo recursivamente.
- Ahora coloco al maestro $M3$ en el grupo $S1$: $S_1 = [M1, M3], \quad S_2 = [M2]$. El coeficiente parcial es totalizando 20. Como 20 es mayor que 14, descarto esta solución. Retrocedo y pruebo al maestro $M3$ en $S2$.
- Coloco al maestro $M3$ en $S2$: $S_1 = [M1], \quad S_2 = [M2, M3]$. El coeficiente parcial es 26. Como 26 es mayor que 14, descarto esta solución. Retrocedo.
- Como al maestro $M2$ ya no lo puedo ubicar en otros grupos retrocedo otro paso más. Lo próximo a hacer sería cambiar de lugar al maestro $M1$ pero como se explicó anteriormente, analizar este espacio de soluciones me daría los mismos resultados.

Finalmente, la mejor distribución encontrada es $S_1 = [M1], \quad S_2 = [M2, M3]$ con un coeficiente total de 14.

Este proceso demuestra cómo el backtracking ayuda a explorar todas las configuraciones posibles de manera eficiente, descartando soluciones parciales no prometedoras sin la necesidad de tener que llegar a completarlas.

El corte que hicimos en el último paso es un corte muy importante ya que nos permite ahorrarnos muchas soluciones para explorar que nos llevarían al mismo resultado, y por lo tanto, mucho tiempo. Pero este corte se puede generalizar.

Supongamos que tenemos 3 maestros con las siguientes habilidades $[2, 2, 1]$ para ordenar en 2 grupos. Una posible solución podría ser $S1 = [2, 1], \quad S2 = [2]$ con un coeficiente de 13. Otra solución posible también podría ser $S1 = [2], \quad S2 = [2, 1]$ con un coeficiente de 13. Si observamos estas soluciones, nos damos cuenta de que ambas llevan al mismo coeficiente y la única diferencia es el intercambio de un maestro entre los grupos.

Este fenómeno ocurre debido a que, en un paso anterior, nos encontramos con los grupos de esta manera: $S1 = [2], \quad S2 = [2]$. En ese punto, las sumas de los grupos son iguales. Podemos deducir que, si dos grupos tienen la misma suma, cualquier decisión de agregar el siguiente número a uno u otro grupo conduciría a soluciones que son, en esencia, idénticas. Por lo tanto, explorar ambas ramas del árbol de búsqueda es redundante.

En otras palabras, si tenemos dos grupos con la misma suma, no es necesario probar agregar el siguiente número a ambos grupos. Hacerlo resultaría en probar casos equivalentes que conducirán a soluciones similares.

Esta misma lógica se puede extender a la partición en k grupos. Si, en algún punto del árbol de búsqueda, encontramos que dos o más de los k grupos tienen la misma suma, podemos evitar explorar todas las posibles combinaciones de asignaciones para estos grupos. En su lugar, se elige explorar solo una de estas combinaciones, ya que las demás serían redundantes. Por ejemplo, si tenemos k grupos y encontramos que tres de ellos tienen la misma suma, solo necesitamos probar asignar el siguiente número a uno de estos tres grupos, ignorando las combinaciones con los otros dos.

Al aplicar esta poda a k grupos, podemos reducir aún más el espacio de búsqueda y mejorar la eficiencia del algoritmo. Esta poda fue de gran utildad dentro del algorítmo ya que redució los tiempos de ejecución signficativamente.

Por último, la forma en la que ubicamos a los maestros en los grupos será de manera que cada maestro se coloque en el grupo con la menor habilidad acumulada hasta el momento. Es decir, la primera opción para ubicar a un maestro será en el grupo que tenga la menor suma de habilidades. Si la suma de este grupo coincide con la de otro grupo donde ya estuvo, se saltará al siguiente grupo con la menor suma acumulada, y así sucesivamente. De esta manera, nos aseguramos de que los maestros se distribuyan equitativamente entre los grupos, minimizando las diferencias de habilidad entre ellos y optimizando la eficiencia de la partición.

Como resumen de la explicación los pasos serían los siguientes:

1. Si ya ubiqué a todos los maestros, evalúo la solución obtenida:
   a) Si el coeficiente es menor que el de la mejor solución encontrada anteriormente, la guardo.
   b) Si no, la descarto.
2. Verifico si la suma del grupo al cual le voy a agregar un maestro es igual a la suma de un grupo ya analizado. Si es así paso al siguiente grupo.
3. Agrego un maestro al grupo
4. Verifico si el coeficiente parcial no supera la mejor solución:
   a) Si el coeficiente parcial no es válido, quito el maestro y vuelvo al paso 2.
   b) Si es válido, llamo recursivamente y vuelvo al paso 1.

3.3. Mediciones Temporales

A continuación se muestran algunas mediciones para distintos tamaños de maestros y cantidades de grupos:

Maestros	Grupos	Tiempo de ejecución [s]
5	2	0.00015
6	4	0.00012
10	3	0.0098
10	10	0.00038
14	3	0.79
14	6	4.26
17	5	292.24
17	7	13.61
18	6	892.68
20	4	3931.63

Cuadro 1: Mediciones para algorítmo de backtracking

4. Solución Propuesta: Algoritmo Programación Lineal

4.1. Implementación

El algoritmo utilizado para obtener las soluciones del problema de la Tribu Agua utilizando Programación Lineal es el siguiente:

```
1  def traducir(groups, maestros):
2      grupos = []
3      for group in groups:
4          grupo = []
5          for i in group:
6              grupo.append(maestros[i])
7          grupos.append(grupo)
8      return grupos
9
10 def p_opt_tribu_agua_pl(maestros, habilidades, k):
11     n = len(habilidades)
12
13     problem = pulp.LpProblem(Diferencia_minima_entre_el_maximo_y_minimo,/
14                                 pulp.LpMinimize)
15
16     x = pulp.LpVariable.dicts("x", ((i, j) for i in range(n)/
17                                 for j in range(k)), 0, 1, cat='Binary')
18     s = pulp.LpVariable.dicts("S", (j for j in range(k)),/
19                                 lowBound=0, cat='Continuous')
20     max_group = pulp.LpVariable("M", lowBound=0, cat='Continuous')
21     min_group = pulp.LpVariable("m", lowBound=0, cat='Continuous')
22
23     problem += max_group - min_group
24
25     for i in range(n):
26         problem += pulp.lpSum(x[i, j] for j in range(k)) == 1
27
28     for j in range(k):
29         problem += s[j] == pulp.lpSum(x[i, j] * habilidades[i]/
30                                                 for i in range(n))
31         problem += max_group >= s[j]
32         problem += min_group <= s[j]
33
34     problem.solve(PULP_CBC_CMD(msg=False))
35
36     groups = [[] for _ in range(k)]
37
38     for i in range(n):
39         for j in range(k):
40             if pulp.value(x[i, j]) == 1:
41                 groups[j].append(i)
42                 break
43
44     sumas = [sum(habilidades[i] for i in group) for group in groups]
45     coeficiente = sum(suma**2 for suma in sumas)
46
47     return traducir(groups, maestros), coeficiente
```

4.2. Explicación del algorítmo

Como primera instancia, la función objetivo que se desea minimizar, corresponde a la suma de los cuadrados de las sumas de las fuerzas/maestrías/habilidades de los grupos, es decir la mostrada en (1).

Como se puede observar, esta función no es lineal, lo que complica el problema, evitando así resolverlo de forma directa.

Las opciones planteadas para resolver esto fueron:

- Obtener la solución óptima
- Minimizar la diferencia del grupo de mayor suma con el de menor suma. De esta forma cambiamos nuestra función objetivo a minimizar a: $\sum_i Z_i - \sum_j Y_j$

En nuestro caso decidimos ir por el segundo caso.

En primera instancia, para avanzar con esta opción planteamos las variables que utilizaremos en nuestro modelo:

- x_{ij}: variables binarias que indican la asignación de maestros a grupos. Si $x_{ij} = 1$, el maestro i está asignado al grupo j. Si $x_{ij} = 0$ el maestro i no pertenece al grupo j.
- S_j: representa la suma de las habilidades de los maestros que peretenecen al grupo j
- max_group: sumatoria de las habilidades del maximo grupo
- min_group: sumatoria de las habilidades del minimo grupo

Por último, la función que vamos a querer minimizar en nuestro problema es: max_group−min_group.

Como segunda instancia, se pasan a definir las restricciones de nuestro problema, para que nuestro algorítmo pueda resolverlo. Hay que destacar que estas restricciones tienen que ser lineales.

Lo primero que tenemos que restringir es que los maestros tienen que pertenecer por lo menos a 1 grupo y como mucho a 1 grupo también. Para esto definimos que:

$$\sum_{j}^{k} x_{ij} = 1 \qquad \forall i \in \text{Maestros} \tag{2}$$

donde k son todos los grupos en los cuales puede estar un maestro.

Para el caso de tener 5 maestros a repartir en 2 grupos nos quedarían las siguientes ecuaciones:

$$x_{11} + x_{12} = 1$$

$$x_{21} + x_{22} = 1$$

$$x_{31} + x_{32} = 1$$

$$x_{41} + x_{42} = 1$$

$$x_{51} + x_{52} = 1$$

La siguiente restricción sera una restricción de igualdad donde estaremos diciendo que el valor de S_j corresponde a la sumatoria de las habilidades de los maestros que pertenecen al grupo j:

$$S_j = \sum_{i}^{n} x_{ij} \cdot h_i \qquad \forall j \in k \tag{3}$$

donde n representa todos los maestros en nuestro probema y h_i es la habilidad del maestro i.

Volviendo al mismo ejemplo ilustrado anteriormente, para el caso de tener 5 maestros para ubicar en 2 grupos quedaría:

$$S_1 = x_{11} \cdot h_1 + x_{21} \cdot h_2$$

$$S_2 = x_{12} \cdot h_1 + x_{22} \cdot h_2$$

Por último nos queda definir las restricciones mediante la cual *obtendremos* el máximo y mínimo grupo. Para eso tenemos que definir que el máximo grupo (grupo con mayor sumatoria en sus habilidades) tiene que ser mayor o igual a todos los grupos y el mínimo grupo (grupo con menor sumatoria en sus habilidades) tiene que ser menor o igual a todos los grupos:

$$\text{max\_group} \geq s[j] \qquad \forall j \in k \tag{4}$$

$$\text{min\_group} \leq s[j] \qquad \forall j \in k \tag{5}$$

Nuevamente para el ejemplo definido anteriormente, para el max_group nos quedaría que:

$$\text{max\_group} \geq S_1$$

$$\text{max\_group} \geq S_2$$

tomando así el valor del S_j con mayor valor.
Para el caso de min_group quedaría:

$$\text{min\_group} \leq S_1$$

$$\text{min\_group} \leq S_2$$

tomando así el valor del S_j con menor valor.
Una vez definidas todas las restricciones al problema utilizamos la libreria `pulp` para obtener que valores tomaran mis variables. De esta forma solo basta con entender los valores de x_{ij} para formar los grupos y una vez formado los grupos realizar la cuenta del coeficiente.

4.3. Mediciones

A continuación se muestran algunas mediciones para distintos tamaños de maestros y cantidades de grupos:

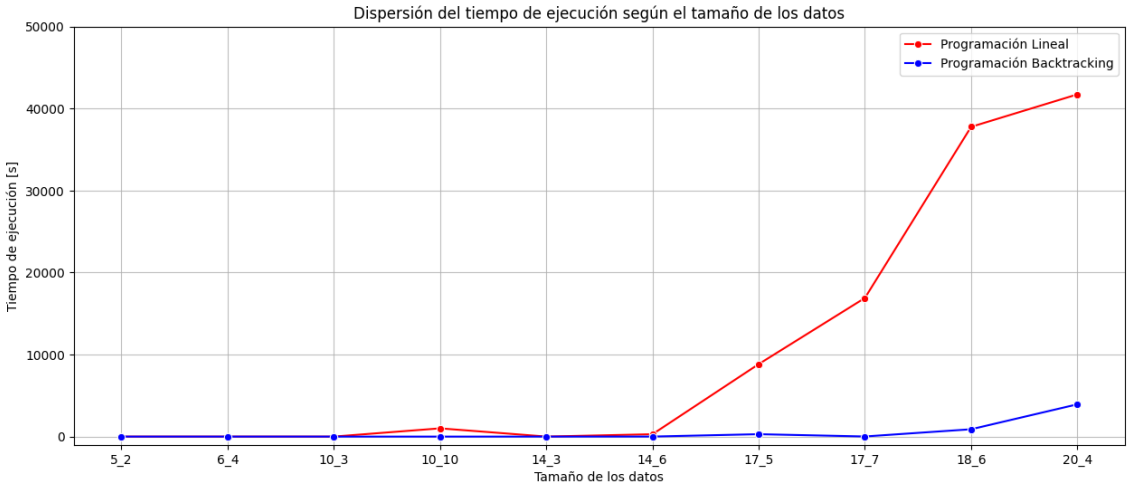
Maestros	Grupos	Tiempo de ejecución [s]
5	2	0.041
6	4	0.11
10	3	0.31
10	10	1003.27
14	3	2.29
14	6	288.16
17	5	8836.34
17	7	16902.91
18	6	37774.26
20	4	41731.87

Cuadro 2: Mediciones para algorítmo de programación lineal

Como se puede observar, los tiempos aumentan a grandes valores a medida que aumentamos tanto la cantidad de maestros como la cantidad de grupos.
Si lo comparamos con el algorítmo de backtracking vemos que obtenemos tiempos mucho más grandes aquí.

Maestros	Grupos	Tiempo de ejecución PL [s]	Tiempo de ejecución BT [s]
5	2	0.041	0.00015
6	4	0.11	0.00012
10	3	0.31	0.0098
10	10	1003.27	0.00038
14	3	2.29	0.79
14	6	288.16	4.26
17	5	8836.34	292.24
17	7	16902.91	13.61
18	6	37774.26	892.68
20	4	41731.87	3931.63

Cuadro 3: Mediciones para algorítmo de programación lineal



5. Solución Propuesta: Algoritmo Greedy

5.1. Algoritmo Greedy del Maestro Pakku

Como alternativa a los algoritmos anteriormente implementados, se nos ha provisto el siguiente algoritmo Greedy:

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

Este se trata de un algoritmo Greedy debido a que se aplica iterativamente una regla sencilla que nos permite obtener el óptimo local en mi estado actual, con la esperanza de encontrar una solución global óptima al final.

A continuación se muestra la implementación del mismo en Python.

```
1  def insertar_ordenado(grupos, grupo_minimo):
2      valores = [grupo[1]**2 for grupo in grupos]
3      valor_minimo = grupo_minimo[1]**2
4
5      posicion = bisect.bisect_left(valores, valor_minimo)
6      grupos.insert(posicion, grupo_minimo)
7
8      return grupos
9
10 def p_opt_tribu_agua_gd(maestros, habilidades, k):
11     tuplas_maestros = list(zip(maestros, habilidades))
12     maestros_ordenados = sorted(tuplas_maestros, key=lambda x: x[1], reverse=True)
13     grupos = [([], 0) for _ in range(k)]
14
15     for maestro in maestros_ordenados:
16         grupo_minimo = grupos.pop(0)
17         nuevo_valor = grupo_minimo[1] + maestro[1]
18         maestros_grupo = grupo_minimo[0] + [maestro[0]]
19         grupo_minimo = (maestros_grupo, nuevo_valor)
20
21         grupos = insertar_ordenado(grupos, grupo_minimo)
22
23     grupos_finales = [x[0] for x in grupos]
24     coeficiente = sum(x[1]**2 for x in grupos)
25
26     return grupos_finales, coeficiente
```

Para mejor entendimiento de la implementación del algoritmo, se explicará en detalle cada sección.

```
24      tuplas_maestros = list(zip(maestros, habilidades))
25      maestros_ordenados = sorted(tuplas_maestros, key=lambda x: x[1], reverse=True)
26      grupos = [([], 0) for _ in range(k)]
```

En primer lugar, se crea una lista la cual contiene a las tuplas (MAESTRO, HABILIDAD), ordenada descendentemente por habilidad. También se inicializa la lista en donde irán los grupos, los cuales estarán representados por las tuplas ([MAESTROS PERTENECIENTES AL GRUPO], SUMA HABILIDADES MAESTROS).

```
26      for maestro in maestros_ordenados:
27          grupo_minimo = grupos.pop(0)
28          nuevo_valor = grupo_minimo[1] + maestro[1]
29          maestros_grupo = grupo_minimo[0] + [maestro[0]]
30          grupo_minimo = (maestros_grupo, nuevo_valor)
31
32          grupos = insertar_ordenado(grupos, grupo_minimo)
33
34      grupos_finales = [x[0] for x in grupos]
35      coeficiente = sum(x[1]**2 for x in grupos)
```

Luego se recorre la lista de maestros ordenados por habilidad, y por cada iteración, se saca de la lista de grupos al grupo con menor valor de habilidad acumulada hasta el momento *grupoMin*. Cabe destacar que en el algoritmo implementado este siempre va a ser el grupo correspondiente a la primera posición, debido a que mantendremos la lista ordenada de manera ascendente por habilidad acumulada. Al *grupoMin* se le agregará el maestro correspondiente a la iteración actual, para a continuación volver a agregar al *grupoMin* a la lista de grupos, de manera ordenada.

```
26  def insertar_ordenado(grupos, grupo_minimo):
27      valores = [grupo[1]**2 for grupo in grupos]
28      valor_minimo = grupo_minimo[1]**2
29
30      posicion = bisect.bisect_left(valores, valor_minimo)
31      grupos.insert(posicion, grupo_minimo)
32
33      return grupos
```

Se utiliza la librería de Python "bisect", debido a que su método `bisect_left` nos permite realizar una búsqueda binaria para encontrar la posición en la que *grupoMin* puede ser insertado en la lista de grupos mientras se mantiene el orden ascendente. Para añadir la menor complejidad temporal posible, se mantentiene la lista de grupos ordenada y se inserta al grupo modificado, utilizando búsqueda binaria para la búsqueda del índice de la posición correcta. La complejidad total de este algoritmo es la resultante de, ordenar a los maestros por habilidades O(m.log(m)), luego, que por cada iteración de los maestros O(m), se reubique al *grupoMin* modificado en la posición correspondiente a la habilidad acumulada O(k), por lo que la complejidad de este algoritmo es O(m.k) + O(m.log(m)).

5.1.1. Cota de Medición Empírica

Este algoritmo sirve como una aproximación para resolver el problema de la Tribu Agua. Pero a priori no se sabe que tanto se aproxima a la solución óptima. Para poder verificarlo, se realizará un análisis empírico para diferentes instancias de problemas aleatorios variando cantidad de maestros, cantidad de grupos y el rango de habilidad que puede tener cada maestro.

En cada una de estas instancias se calcula lo siguiente. Sea I una instancia cualquiera, y $z(I)$ una solución óptima para dicha instancia, y sea $A(I)$ la solución aproximada, se define:

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

Para todas las instancias posibles.

A partir del valor de las relaciones, se puede inferir que tanto se aleja de la solución óptima. Se realizaron mediciones para 50 instancias aleatorias con una cantidad menor o igual a 12 maestros con

un rango de habilidad de 1 a 1000 y con la cantidad de grupos constante. Para cada instancia, se corre el algoritmo Greedy y el algoritmo de backtracking que obtiene la solución óptima. Luego se divide el resultado aproximando entre el óptimo y se toma ese punto del gráfico.

Como se puede apreciar, el algoritmo tiene una cota de error cercana al 0.0025 lo que indica que el algoritmo Greedy tiene un error aproximado al 0.3 %. Aunque no siempre obtiene el resultado correcto, sigue siendo útil pues da un valor aproximado en un tiempo considerablemente menor.

Cabe aclarar que cada gráfico tiene 50 instancias de problemas diferentes. Por lo tanto no se pueden comparar entre sí. Pero el objetivo es visualizar como se comporta el algoritmo en diferentes problemas para obtener empíricamente el valor de cota de error.

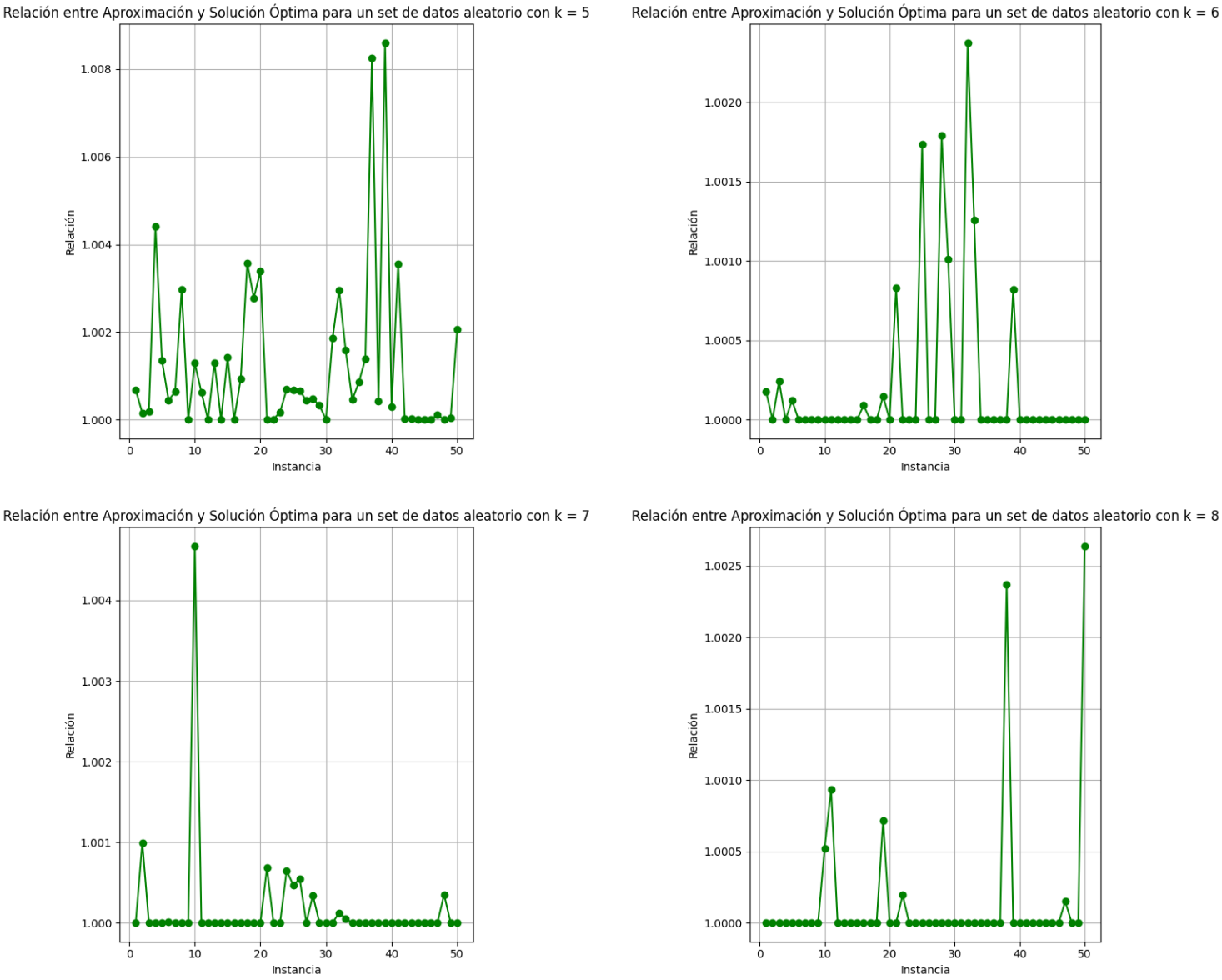
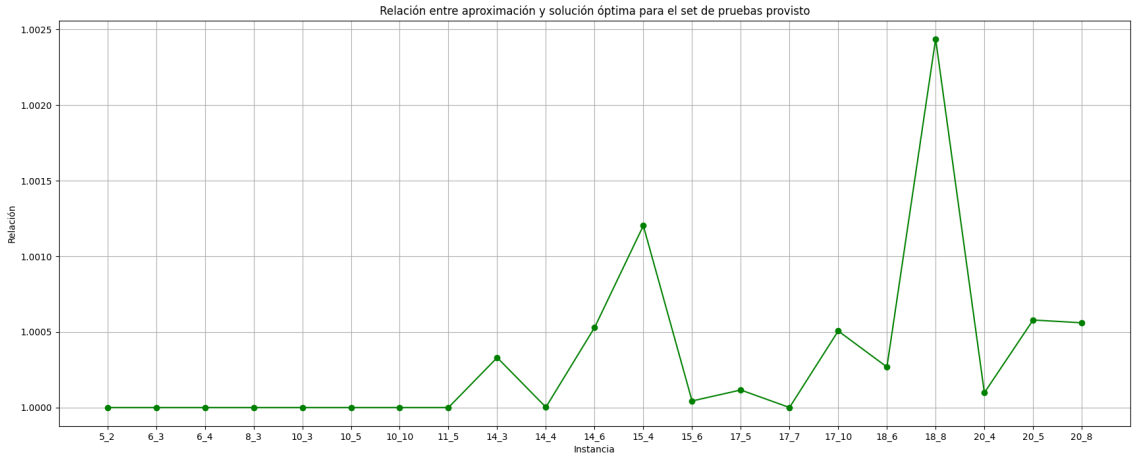


Figura 1: Comparación de instancias aleatorias variando la cantidad de subgrupos para el algoritmo greedy del Maestro Pakku

Además, se nos ha provisto un set de pruebas para distintos valores de maestros y grupos. Graficando la misma relación entre el resultado aproximado y el resultado óptimo (El cual es conocido sin necesidad de calcularlo) se llega a resultados similares. El siguiente gráfico indica la instancia correspondiente con el formato {Cantidad de maestros}_{Cantidad de grupos} y su valor de relación con la solución aproximada del algoritmo Greedy propuesto.



Se puede apreciar que se llega a una cota de error similar a las corridas aleatorias anteriores incluso en instancias de problemas que demoran mucho en resolverse con Backtracking o Programación Lineal.

5.2. Algoritmo Greedy Alternativo

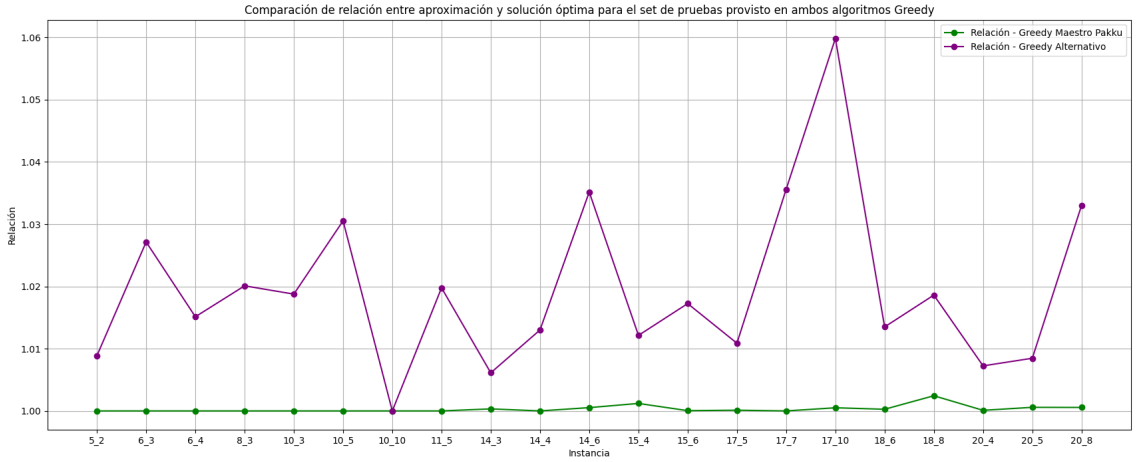
Para completar el análisis, se analizará otra estrategia Greedy. El algoritmo es el siguiente: *Calcular la suma total de las habilidades de todos los maestros y dividirla por la cantidad de grupos, dando una SumaHabilidadIdeal. Luego, Ordenar de mayor a menor los maestros en función de su habilidad o fortaleza. Iterar los grupos de manera circular (si se llega al último grupo, se comienzan a iterar nuevamente desde el principio) e ir colocando un maestro por cada iteración. Si la suma acumulada de fuerza de un grupo supera la SumaHabilidadIdeal, se saltea.*

```
1  def p_opt_tribu_agua_gd2(maestros, habilidades, k):
2      suma_habilidades = sum(habilidades)
3      habilidades_optimo = suma_habilidades / k
4      tuplas_maestros = list(zip(maestros, habilidades))
5      maestros_ordenados = sorted(tuplas_maestros, key=lambda x: x[1], reverse=True)
6
7      grupos = [[] for _ in range(k)]
8      suma_grupos = [0] * k
9      indice_grupo = 0
10
11     for maestro in maestros_ordenados:
12         indice_grupo = indice_grupo % k
13         while suma_grupos[indice_grupo] > habilidades_optimo:
14             indice_grupo += 1
15         grupos[indice_grupo].append(maestro[0])
16         suma_grupos[indice_grupo] += maestro[1]
17         indice_grupo += 1
18
19     coeficiente = sum(x**2 for x in suma_grupos)
20
21     return grupos, coeficiente
```

La complejidad de este algoritmo se calcula tomando en cuenta que:

- El ordenamiento de maestros a partir de su habilidad O(m.log(m))
- Por cada iteración de un maestro, se recorren los grupos hasta encontrar un grupo que no haya pasado la *SumaHabilidadIdeal* (cota superior O(k)), lo que resulta en O(m.k). Como resultado se tiene que la complejidad es O(m.log(m)) + O(m.k).

Haciendo un análisis análogo al algoritmo Greedy del Maestro Pakku, se obtuvo empíricamente que la relación R(A) para el algoritmo alternativo es de 1.15, lo cual indica un performance mucho más pobre que la del primer algoritmo Greedy.



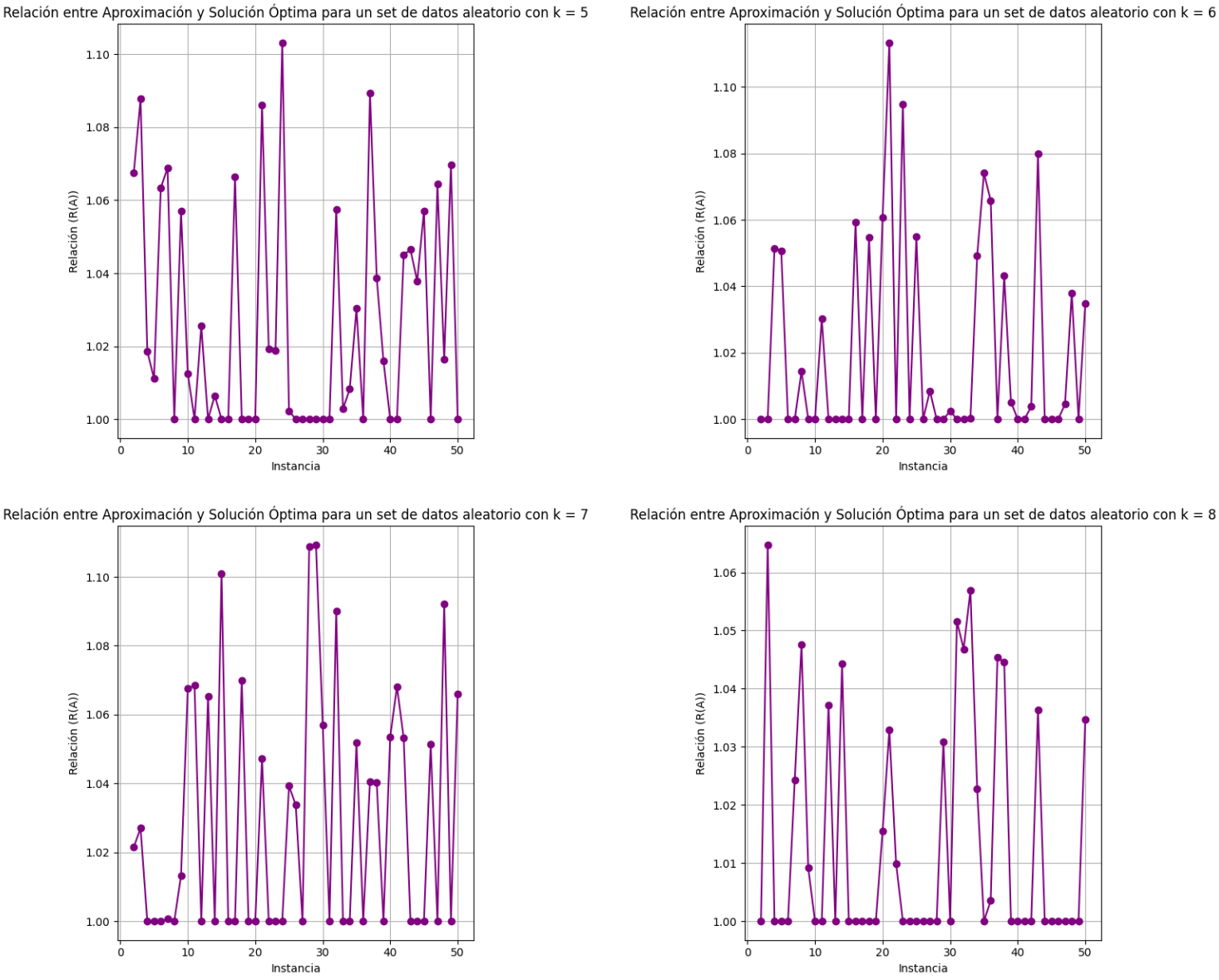
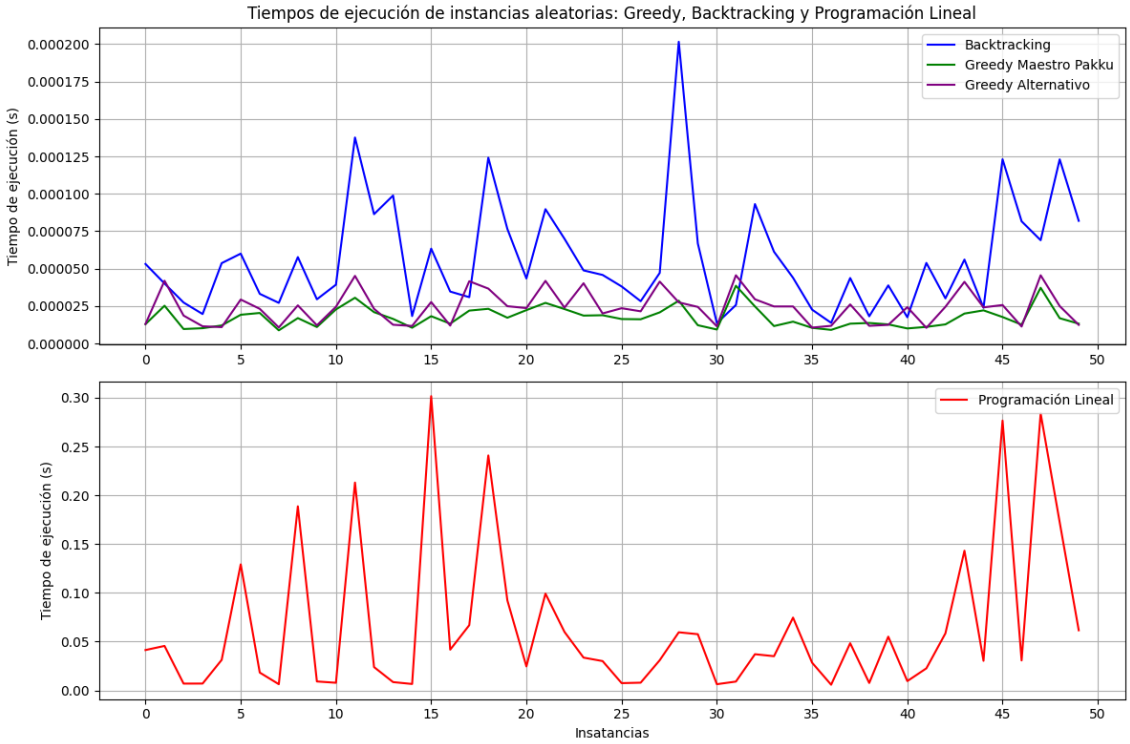


Figura 2: Comparación de instancias aleatorias variando la cantidad de subgrupos para el algoritmo greedy alternativo

6. Conclusiones y Comparación de resultados: Backtracking, Programación Lineal y Greedy

Para concluir con el análisis del problema, se realizará una comparación del rendimiento temporal de los problemas.

Para ello se generaron instancias aleatorias de problemas de manera similar a los incisos anteriores. Se corrieron 50 problemas diferentes con un máximo de 6 maestros, un rango de habilidades entre 1 y 1000 y una cantidad de conjuntos entre 2 y 5.



De este gráfico se pueden inferir las siguientes conclusiones:

- El algoritmo de programación lineal es, con diferencia, el que más tarda encontrar el óptimo. Justamente por eso se decidió graficarlo por aparte para no entorpecer la escala.
- El algoritmo de Backtracking es el que más tarda después del de Programación Lineal. Si bien por lo general demora más que ambos algoritmos Greedy, a diferencia de ellos si obtiene la solución óptima.
- El primer algoritmo Greedy posee una cota de error máxima empírica de 0.3 %.
- El segundo algoritmo Greedy generado para conseguir una aproximación de la solución del problema posee una cota de error máxima empírica de 15 %, bastante más pobre que la primera, debido a que no posee en cuenta qué grupo posee la menos habilidad acumulada.

Algunas conclusiones que se obtuvieron a partir de la realización del trabajo práctico:

- Con respecto al algoritmo de backtracking, fue crucial tener un entendimiento claro del problema y de cómo se conforma la solución para obtener buenas podas.
- Observamos que una poda bien implementada puede ayudar significativamente a resolver el problema de manera eficiente, reduciendo notablemente los tiempos de ejecución sin dejar de encontrar la solución óptima.
- Al desarrollar el algoritmo de programación lineal, notamos una mayor facilidad en comparación con otras estrategias al abordar el problema. Esto facilitó la implementación de nuestro algoritmo debido a su estrctura más directa y menos dependiente de algunas condiciones

- Sin embargo, desde una perspectiva negativa, el algoritmo resultó ser menos eficiente en términos de tiempo, ya que observamos un aumento significativo en el tiempo de ejecución en comparación con otros algoritmos al resolver los mismos casos.
- Si hablamos de la escabilidad de estos algorítmos, podemos observar que la progrmación lineal y el backtracking se mantienen estables y con tiempos muy similares en problemas de menor tamaño. Mientras que, a diferencia de programación lineal, el algorítmo de backtracking se mantiene eficiente en problemas más grandes implementando buenas podas.
- Si bien el primer algoritmo Greedy no garantiza el resultado óptimo del problema se comprobó que el error que se obtiene al utilizar el mismo es mínimo y que, en contraposición, el tiempo que se ahorra debido a la reducción de complejidad del problema es altísima.