Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

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

14 de junio de 2024

Leticia Figueroa
110510

Andrea Figueroa
110450

Josue Martel
110696

1. Introducción

El objetivo del trabajo práctico es ayudar a la Tribu del Agua en pro de defenderse del ataque de la nación del fuego. Para ello buscamos seguir las recomendaciones del maestro Pakku: mediante una estrategia de distribución de los maestros lograr mantener un ataque constante.

En el presente trabajo buscamos encarar el problema de la Tribu de Agua a través de: el análisis de las características y naturaleza del problema (¿Podemos clasificarlo?) para posteriormente lograr construir una propuesta de una solución por búsqueda exhaustiva al problema planteado (versión de optimización), una propuesta de solución mediante un modelo de programación lineal, así como una aproximación greedy a la solución óptima del mimso, realizar mediciones a los resultados obtenidos para finalmente poder concluir qué heurística resultó valiosa para la resolución del problema planteado.

Por lo que, respecto a las propuestas que mencionamos que buscamos alcanzar, en el presente informe detallaremos el análisis previo a la concretizacion de estas, la construcción de las mismas y finalmente la implementación de las mismas (incluyendo la revisión de su complejidad).

2. Análisis del problema: conceptos básicos, terminología y entidades del problema

A continuación, presentaremos el análisis inicial del problema, seguido de una clasificación del mismo. Incluiremos, además, la correspondiente demostración de la categoría previamente establecida. Para comenzar, planteamos desde ahora las siguientes convenciones:

- Cantidad total de maestros Agua: n
- Fuerza/skill del maestro Agua i: x_i
- Conjunto total de maestros: S
- La cantidad de particiones/subgrupos: k
- Los subgrupos de maestros $(S_1, S_2, ..., S_k)$

De la misma forma, la terminología que usaremos al referirnos a las diferentes formas en las que se puede abordar la situación planteada se describe a continuación:

Al decir **"la versión de optimización"** nos referimos a nuestro problema original, donde buscamos encontrar la mejor solución entre las que cumplen con el criterio establecido. Es decir, la versión en la que buscamos minimizar la suma de los cuadrados de las fuerzas de los grupos:

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

Por otro lado, al decir **"la versión de decisión"** nos referiremos a la instancia del problema en la que buscamos decidir (valga la redundancia) si existe una solución para nuestro problema dado los datos de entrada. Es decir, dados dos números k y B, buscamos definir si existe una partición en k grupos $S_1, S_2, ..., S_k$ tal que se cumpla:

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

Adicionalmente, en ocasiones denominaremos al Problema de la Tribu del Agua como P.T.A para abreviar.

El primer análisis a realizar se trata de la naturaleza de la versión de decisión:

3. Clase de complejidad:

3.1. ¿NP?

Para los fines del siguiente análisis, diremos que un problema pertenece a la clasificación NP si existe un certificador eficiente para el mismo. En nuestro caso, nuestro certificador eficiente plantea un algoritmo polinomial que, dada una entrada (como la partición que se busca obtener en el problema de optimización y un entero "b"), busca que se cumpla el criterio para la versión de decisión. Es decir, para verificar que la **versión de decisión** del problema se encuentra en NP, planteamos un algoritmo que suma la suma de las habilidades de cada maestro de una partición elevada al cuadrado.

Desde ya, podemos intuir por qué afirmamos que los pasos que realiza nuestro certificador son polinomiales. Básicamente, sumamos todas las habilidades de los maestros a disposición ($O(n)$, siendo n la cantidad de maestros totales), donde, si estaban agrupados en una misma partición, se les realizó una operación matemática ($O(1)$). Como realizar dicha operación matemática no nos consume tiempo computacional en función de la cantidad de maestros de nuestro problema, la complejidad del certificador propuesto es lineal en función de la cantidad de maestros, es decir, es un certificador eficiente. Veamos a continuación el algoritmo en cuestión:

```
1     def effective_validator(partitions,b ):
2         skills = 0
3         for partition in partitions:
4             skills_group = 0
5             for _,skill in partition:
6                 skills_group += skill
7             skills += (skills_group**2)
8             if skills > b:
9                 return False
10        return True
```

Podemos observar en la implementación planteada que, efectivamente la complejidad de nuestro validador resulta uno polinomial:

$$O(n) \tag{3}$$

3.2. ¿NP-Completo?

Comenzaremos aclarando los principios y conceptos esenciales en los que nos basamos para hacer las siguientes afirmaciones.

- Reducir polinomialmente un problema X a otro Y significa que se puede transformar toda instancia del problema X, a través de una cantidad polinomial de pasos, a una instancia del problena Y
- Usaremos la notación $X \leq_p Y$ para indicar que el problema X se puede reducir polinomialmente a Y.
- Si $X \leq_p Y$, entonces el problema Y es al menos tan complicado como X (puede ser más complicado)
- Si logramos reducir un problema NP-Completo a un problema X, sostenemos que X debe ser NP-Completo (y esto nos permitirá realizar las demostraciones siguientes)

Además, dos problemas en su versión de decisión formarán parte de nuestra estrategia para demostrar que el problema de la Tribu del Agua al que nos enfrentamos pertenece efectivamente al conjunto NP-Completo. Estos problemas son: 2-partition y Subset Sum.

3.2.1. Estrategia de reducción

Para llevar a cabo la reducción de un problema NP-Completo al *problema de la Tribu del Agua*, buscaremos demostrar que el segundo es al menos tan difícil como el problema *2-partition*, y que este mismo es al menos tan difícil como el problema *Subset sum* (el cual ya se mostró peteneciente al conjunto de los problemas NP-Completos durante la cursada). Es decir:

$$SubsetSum \leq_p 2 - partition \leq_p Problema\ de\ la\ Tribu\ de\ Agua \tag{4}$$

Como la propiedad $\leq_p$ es transitiva, si logramos esa serie de reducciones entonces habremos logrado reducir un problema NP-Completo conocido: *Subset sum* a otro problema *2-Partition*, con lo cual este resulta también NP-Completo, y este mismo reduciremos al *problema de la Tribu del Agua*. Es entonces que podremos afirmar que nuestro problema es efectivamente NP-Completo.

Comencemos estableciendo en qué consiste el nuevo problema mecionado:

> **2-partition:** Dado un conjunto M de n números enteros ($M = m_1, m_2, ..., m_n$). ¿Es posible obtener una partición de dos subconjuntos de M: M_1 y M_2 tales que la suma de los elementos en M_1 sea igual a la suma de los elementos en M_2?.

3.2.2. 2-partition: problema NP-Completo

Como ya establecimos previamente, para un problema ser considerado como NP-Completo primero debemos demostrar que este perenece a NP. A continuación expondremos nuestro planteamiento de verificador eficiente para el problema 2-partition:

2-Partition pertenece a NP:

> Dado un subconjunto M y dos subconjuntos M_1 y M_2, la verificacion de que estos últimos sean particiones de M se logra en tiempo polinomial sumando los elementos de M_1 así como los de M_2 y verificando que estos dos resultados sean iguales. Entonces, si el total de los elementos del conjunto M son n, entonces la complejidad resulta $O(n)$.

Además, si encontramos una reducción de un problema NP-Completo (en su version de decision) al problema 2-partition (también en su version de decisión), demostramos que 2-partition es también un problema NP-Completo:

Reduccion de Subset Sum a 2-partitition:

> *-Recordemos el problema Subset Sum en su versión de decisión: consiste en decidir si existe un subconjunto dentro de un conjuto M dado de enteros cuya suma sea exáctamente igual a un número dado T.*
>
> ***Nota:** En lo siguiente denominaremos $Sum(X)$ a la suma de los elementos que conforman el conjunto X.*
>
> Entonces, para llevar a cabo la reducción de Subset Sum a 2-Partition:
>
> 1. Transformaremos la instancia del problema Subset Sum en una del problema 2-Partition definiendo el conjunto M a partir de los elementos del conjunto S agregando un nuevo elemento $s_0 = (2T) - Sum(S)$, entonces $M = S \cup \{s_0\}$
> 2. Con esta regla de equivalencia entre M y S ya podemos establecer que existe un subconjunto $S_1 \subseteq S$ tal que la suma de sus elementos sea T sii existe una particion de M de dos subconjuntos que sumen lo mismo.

- **Si hay Subset Sum del conjunto S que sume T, entonces hay un 2-Partition del conjunto M:**

  Observación 1.- Si S_1 es un subconjunto de S entonces podemos agrupar aquellos elementos de S que no estén en S_1 en un subconjunto, llámese S_2, con lo que tenemos definida una partición de S.

  Observación 2.- $Sum(S) = Sum(S_1) + Sum(S_2)$ y como sabemos la suma de elementos del subconjunto S_1, entonces $Sum(S_2) = Sum(S) - T$.

  Sabiendo que $M = S \cup \{s_0\}$, podemos escribir a S como la unión de la partición que ya definimos: $M = S_1 \cup S_2 \cup \{s_0\}$, entonces si denominamos: $M_1 = S_1$ y $M_2 = S_2 \cup \{s_0\}$ ya tenemos la partición en dos subconjuntos tal que la suma de ambos es igual, pues:

  - $M = M_1 \cup M_2$
  - $Sum(M_1) = Sum(S_1) = T$
  - $Sum(M_2) = Sum(S_2) + 2T - Sum(S) = Sum(S) - T + 2T - Sum(S) = T$

- **Si hay un 2-Partition del conjunto M entonces hay Subset Sum del conjunto S que sume T:**

  Observación 3.- Si $M = S \cup \{s_0\}$ entonces podemos escribir una partición cualquiera de M en dos subconjuntos (M_1 y M_2) partiendo de una partición de S en dos subconjuntos (S_1 y S_2) pero agregando el elemento faltante s_0 en alguno de estos subconjuntos, sin perder generalidad, digamos que $M_1 = S_1$ y que $M_2 = S_2 \cup \{s_0\}$

  Observación 4.- Basándonos en lo propuesto en la anterior observación, entonces el valor de la suma de los elementos en el subconjunto M_2 sería $Sum(M_2) = Sum(S_2) + 2T - Sum(S) = Sum(S_2) + 2T - Sum(S_1) - Sum(S_2)$ con lo que podemos establecer que $Sum(M_2) = 2T - Sum(S_1)$

  Sabiendo que $M = M_1 \cup M_2$ es una partición de M y que, por hipótesis, existe una partición de dos conjuntos tal que sus sumas sean iguales, entonces podemos plantear la igualdad $Sum(M_1) = Sum(M_2)$ con lo que $Sum(S_1) = 2T - Sum(S_1)$ Y finalmente: $Sum(S_1) = T$

3.2.3. 2-partition $\leq_p$ Problema de la Tribu del Agua

Eventualmente usaremos las siguientes convenciones para referirnos a ciertas características de este problema:

- Cantidad de elementos: n
- Suma total de los elementos de M: $N = \sum_{i=1}^{n} m_i$
- Suma total de los elementos de la partición M_1: $A_2 = \sum_{j=1} m_i$
- Suma total de los elementos de la partición M_2: A_1

Buscamos que la versión de decisión del problema de la Tribu del Agua nos diga si se puede resolver 2-partition.

El P.T.A evalua si existe partición tal que:

$$\sum_{i=1}^{k} \left(\sum_{x_j \in S_i} x_j \right)^2 \leq B \tag{5}$$

Para reducir el problema de 2-partition establecemos las equivalencias y transformaciones que requeriremos para adaptarlo al P.T.A. En principio, la cantidad de particiones será fija: $k = 2$, entonces 5 se puede desglosar como:

$$\left(\sum_{m_j \in M_1} m_j\right)^2 + \left(\sum_{m_i \in M_2} m_i\right)^2 \leq B \tag{6}$$

Que por convenciones previas, podemos expresar como:

$$A_1^2 + A_2^2 \leq B \tag{7}$$

Además, como A_1 y A_2 son conjuntos disconjuntos, entonces diremos que $A_2 = N - A_1$. Entonces la ecuación anterior (7) se puede reescribir como:

$$A_1^2 + (N - A_1)^2 \leq B \tag{8}$$

Ahora bien, queremos estudiar qué sucede si es que A_1 es exactamente $\frac{N}{2}$. Es fácil observar que, remplazando en la ecuación 8, podemos afirmar que:

$$(\frac{N}{2})^2 + (\frac{N}{2})^2 = \frac{N^2}{2} \tag{9}$$

Concretamente, lo antes explicado pretende expresar lo siguiente:

$$\text{si } A_1 = \frac{N}{2} \implies A_1^2 + A_2^2 = \frac{N^2}{2} \leq B \tag{10}$$

Ahora bien, necesitamos probar la doble implicancia para poder concluir nuestra reducción. Partiendo del supuesto de que $A_1^2 + A_2^2 \leq \frac{N^2}{2}$ queremos ver que esto implica que $A_1 = \frac{N}{2}$. Veamos:

$$A_1^2 + (N - A)^2 \leq \frac{N^2}{2} \tag{11}$$

donde, desglozando el lado izquierdo de la desigualdad, esta es equivalente a:

$$A_1^2 + A_1^2 - 2 \times A_1 \times N + N^2 \leq 2 \times A_1^2 - 2 \times A_1 \times N^2$$

Ahora, si retomamos en la ecuación previa:

$$2 \times A_1^2 - 2 \times A_1 \times N + N^2 \leq \frac{N^2}{2} \tag{12}$$

Si analizamos el lado izquierdo de la desigualdad, y sabiendo que para nuestro problema N es fijo y lo que nos interesa es A_1 (Es decir, las particiones). Podemos buscar la cota de la siguiente función a traves de un proceso de optimización. Concretamente, si:

$$f(A_1) = 2 \times A_1^2 - 2 \times A_1 \times N + N^2$$

Dicha función solo tiene un punto crítico:

$$\frac{df(A_1)}{dA_1} = 4 \times A_1 - 2 \times N = \frac{N}{2}$$

se puede mostrar que al $f(A_1)$ ser una parábola positiva, el punto crítico encontrado corresponde a un mínimo absoluto,este se alcanza en $A_1 = \frac{N}{2}$ y toma el valor de $f(A_1 = \frac{N}{2}) = \frac{N}{2}$.

Retomando en la ecuación 11, al ser $A_1 = \frac{N}{2}$ el valor para A_1 donde se alcanza la cota establecida y como este tiene la característica de ser un mínimo absoluto podemos **concretizar** la información diciendo que:

$$\text{si } A_1^2 + A_2^2 \leq \frac{N^2}{2} \implies A_1 = \frac{N}{2}, A_2 = \frac{N}{2} \tag{13}$$

Entonces **concluimos** que si a la entrada del P.T.A le ingresamos las siguientes transformaciones de la entrada del problema 2-partition (que es NP-Completo), entonces cuando el problema de la tribu del agua tenga solución para esos valores de entrada el problema 2-partition también la tendrá y cuando P.T.A no la tenga, entonces 2-partition tampoco la tendra: Entonces el Problema de la Tribu del Agua es al menos tan dificil que un problema NP-Completo, es decir que este tambien es NP-Completo. Respecto a las transformaciones y la entrada del problema de la Tribu del Agua:

- cantidad de subgrupos/particiones = 2
- habilidades de los maestros = conjunto de numeros enteros (universo)
- Cota (B) = $\frac{N}{2}$ (siendo N la suma de todos los números enteros del universo)

4. Construcción de la solución por backtracking

Habiendo demostrado la NP-Complitud de nuestro problema, suena razonable plantear un algoritmo que explore exhasutivamente las factibles soluciones de este.

4.1. Generación de las posibles soluciones al problema

La optimización prematura es la raíz de todo mal.

Primero lo primero. En el desarrollo de nuestro algoritmo por búsqueda exhaustiva para este problema particular, de igual forma que para el desarrollo de otros algoritmos por backtracking, resulta importante el que, antes de pensar en qué posibles optimizaciones o podas se podrían llevar a cabo para poder ahorrar el recorrer caminos que no dirigen a una solución óptima, **primero comprendamos el espacio de soluciones y cómo lograríamos generar todas las configuraciones que podrían llevar a una solución**.

Con lo cual, en la presente sección nos abocaremos a desmenuzar y entender correctamente cómo luciría el espacio de soluciones del problema:

- El espacio de soluciones debe poder representar a cada uno de los n maestros que conformarán los grupos. Por lo cual proponemos la notación M_i para representar el i-avo maestro (por ahora abstrayéndonos de cual es el orden en el que analizamos a los maestros).
- Además de el considerar los n maestros agua, para que la modelación de nuestro espacio de soluciones se asemeje a lo que quisiesemos visualizar en la solución óptima resulta importante a qué grupo se le asigna a cada uno de los n maestros, siendo que cada uno de ellos puede estar en solo uno de entre k opciones.

Con lo cual, estructuramos la información relevante mencionada a través del siguiente vector: $(M_1, M_2, ..., M_n)$, donde el i-avo elemento tiene como valor el número de grupo al que pertenece el i-avo maestro: M_i, por lo tanto los elementos del conjunto:

$$\{(M_1, M_2, ..., M_n) : M_i \in \{1, 2, ..., k\} \ \forall i = 1, 2, ..., n\}$$

representarían todas las cofiguraciones posibles, que de ser el caso en que implementásemos un algoritmo por fuerza bruta, exploraríamos en su totalidad, mas, al estar implementando un algoritmo por backtracking, buscamos optimizar la forma en la que buscamos la mejor solución evitando explorar algunas opciones que sabemos no serán óptimas. Precisamente al análisis de lo recién mencionado nos avocaremos en la siguiente sección (*Vease: 4.2*)

Retomando el tópico de esta sección, entonces el esquema básico que seguiremos para explorar nuestras opciones son: Ir maestro por maestro asociándolo a un grupo, donde las opciones respecto al maestro i son los k grupos a los que puede pertenecer.

4.2. Podas y optimizaciones

En general, el P.T.A tiene justamente como problema las pocas podas que podemos aplicar, de forma que las reducciones en tiempo de ejecuión más significativas se encuentran en las optimizaciones. De cualquier forma, a continuación presentaremos las técnicas que empleamos para reducir el tiempo de ejecución de nuestro algoritmo por búsqueda exhaustiva.

4.2.1. Pseudo óptimo de la función objetivo

La siguiente poda, se basa en el establecimiento de un límite o un estandar básico que cualquier propuesta a solución óptima debe cumplir. En principio, esta cota sería la representación del infinito que el lenguaje proporciona, misma que a medida que vamos explorando las posibles soluciones se iría calibrando.

A continuación desglozaremos el proceso de calibración de este pseudo óptimo y cómo es que podemos partir de una mejor estimación que la representada por el infinito inicial: una primera aproximación.

Primera aproximación Dado que implementar un algoritmo desde cero (es decir, sin una primera aproximación que nos ayude a acotar las soluciones que analicemos) para lograr obtener la solución óptima al problema resulta costoso por la naturaleza del mismo, el primer paso que decidimos tomar es buscar una primera aproximación en la cual basarnos (de esta sabemos que el resultado óptimo será mejor o igual, pero nunca peor). Debido a que en otra sección del informe ya realizamos un análisis profundo del aproximador usado (*propuesta del maestro Pakku*), en esta sección solo especificamos cómo esta misma se puede usar para generar una poda temprana: todos los resultados que sepamos son subóptimos en contraste al ofrecido por la primera aproximación greedy pueden ser descartados.

Con lo que, de la aproximación greedy implementada por el grupo, determinamos que se use el coeficiente (valor de la función objetivo a minimizar) que esta corresponde para compararlo con el coeficiente en construcción de cada posible solución. Por lo que todo "postulante" a óptimo debe poseer un coeficiente menor o igual al obtenido por el algoritmo greedy. Además, cabe mencionar desde ya, que el incorporar esta aproximación al algoritmo de backtracking no conlleva un costo significativo en terminos de complejidad temporal, pues el obtener esta aproximación tiene una complejidad temporal teórica tal que se opaca por la del algoritmo mismo de backtracking, pues la naturaleza de la búsqueda exhaustiva subyacente de su lógica basada en la fuerza bruta termina opacando cualquier otra complejidad polinomial.

Nota: Para mayor detalle respecto a la aproximación a la que hacemos referencia, véase la sección 5)

Calibración del pseudo óptimo: Habiendo partido del pseudo-óptimo ofrecido por la aproximación greedy, es posible que, a medida que exploremos otras soluciones, nos encontremos con mejores cotas. Para ello, decidimos tomar el hallazgo de un mejor resultado que el óptimo previo como el punto en el que se establece una nueva cota. Con la misma lógica con la que establecimos la primera poda respecto al coeficiente de una posible solución, ahora estaremos actualizando constantemente el valor con el cual se compararán las próximas soluciones. Para ser considerada una mejor solución, el coeficiente de la función objetivo debe ser menor al previamente establecido, si esto ocurre, dicho coeficiente será el nuevo estándar para las próximas exploraciones de posibles soluciones.

4.2.2. Optimización previa a la exploración exhaustiva: Ordenamiento de los maestros por habilidad

Basándonos en una regla simple (la misma en la cual se basa el algoritmo de aproximación greedy detallado en la sección 5), la optimización a la cual hacemos referencia incorpora a los maestros con mayor fuerza antes que los demás, reduciendo significativamente el espacio de búsqueda. La afirmación anterior se debe a que, si desde el inicio colocamos a los maestros que podrían generar más conflictos a futuro por su gran capacidad/fuerza (que potencialmente .ªrruinaría"la paridad que poseía una combinación antes de agregar a dicho maestro), evitamos encontrarnos con casos desfavorables que no conducirían a una solución óptima.

Es decir, con un ordenamiento previo logramos que la búsqueda exhaustiva analice a los maestros potencialmente críticos lo más pronto posible, permitiéndonos identificar y podar ramas del árbol de búsqueda que no conducirán a soluciones viables en el menor nivel de profundidad en la recursión en el que se puedan identificar.

4.2.3. Podas incorporadas

Cota al grupo más dispar: Si buscamos la fuerza conjunta del grupo más dispar de una posible solución óptima, sabemos que cualquier otra solución que tuviese alguna oportunidad real a ser la mejor solución debe minimizar esta disparidad o igualarla. Con lo que, decidimos usar el dato del máximo total de fuerza grupal de un solo grupo (de entre las fuerzas conjuntas de cada grupo) para descartar a las soluciones que distribuyen una combinación con un grupo que acumule más habilidad/fuerza dentro de la configuración de uno de sus grupos.

Calibración de la cota: De la misma forma en que hemos estado ajustando las cotas establecidas previamente a medida que obtenemos más información, aplicaremos la misma técnica con la cota actual. Así, a medida que exploramos más opciones y encontramos soluciones que establezcan una cota más precisa, podemos afinar la poda propuesta, evitando así explorar soluciones subóptimas.

4.3. Código

En esta sección, veremos la implementación del esquema general antes explicado. Para no marear al lector, procederemos exponiendo primero las funciones auxiliares involucradas en nuestra búsqueda exhaustiva, precediendolas de una breve descripción de su funcionalidad (obviamos las complejidades puesto que el algoritmo en cuestión donde estas se usarán no tiene una complejidad polinomial).

4.3.1. Funciones auxiliares

1. Obtiene el coeficiente representativo de una combinación dada de los maestros:

```
    def get_coefficient(groups):
        sum_sqr_groups = 0
        for group in groups:
            sum_group = 0
            for bender in group:
                sum_group += bender[1]
            sum_sqr_groups += sum_group ** 2
        return sum_sqr_groups

```

2. Dado un grupo de maestros, obtiene la suma de sus habilidades:

```
    def get_group_sum(group):
        return sum([bender[1] for bender in group])

```

3. Obtiene la suma de habilidades del grupo de maestros con más habilidad conjunta (el grupo más dispar):

```
    def get_max_group_sum(groups):
        max_sum = 0
        for group in groups:
            group_sum = get_group_sum(group)
            if group_sum > max_sum:
                max_sum = group_sum
        return max_sum

```

4. Determina si una solución parcial puede ser descartada debido a que empeora la paridad en los grupos (hay un grupo que acumula más habilidad de la que deberia para seguir siendo una propuesta viable de solución óptima).

```
    def has_a_greater_group_sum(partial_result, max_group_sum):
        for group in partial_result:
            if get_group_sum(group) > max_group_sum:
                return True
        return False

```

4.3.2. Funciones principales

A continuación observaremos el cuerpo de nuestro algoritmo por backtracking, y cómo es que implementamos el uso de las podas antes mencionas y las optimizaciónes que logramos realizar:

Función envolvente: En esta sección aplicamos el ordenamiento antes mnecionado, creamos las estructuras para almacenar una propuesta de solución (que se irá construyendo), realiza el primer llamado recursivo: proveyendola del pseudo óptimo inicial y la cota al grupo más dispar (obtenidas de una aproximación voráz) o en caso de no tener una aproximación previa, llamando a la función recursiva con infinitos coherentes a su uso.

```
    def backtracking_algorithm(benders_skills, groups_count, min_coefficient = math.inf,max_group_sum = math.inf):
        benders_skills = sorted(benders_skills, key=lambda x : x[1], reverse=True)
        groups_result = []
        for _ in range(groups_count):
            groups_result.append([])
        coefficient, _ = group_rec(benders_skills, groups_count, 0, [], groups_result, min_coefficient, max_group_sum)
        return coefficient, groups_result
```

Función recursiva: Comenzamos haciendo el backtrack si la configuración actual ya no es una propuesta viable como solución óptima. Posteriormente, evaluamos si encontramos una mejor solución que la mejor anterior; en ese caso, actualizamos el coeficiente que se usa como cota y regresamos en busca de otras opciones. Antes de comenzar a evaluar las opciones disponibles, realizamos el backtrack si no tenemos opciones para intentar o si el coeficiente de la solución parcial ya es peor que el que debería ser.

El maestro actual puede ser asignado a cualquiera de los grupos ya creados, y exploramos recursivamente la asignación del siguiente maestro. Al regresar de las subramas, quitamos al maestro del grupo al que se había agregado en la configuración (solución parcial) y probamos agregándolo a otro grupo, repitiendo el proceso hasta haber intentado con todos los subgrupos existentes.

Si la solución parcial aún no tiene las particiones requeridas, intentamos crear una nueva que inicia con la asignación del maestro actual y exploramos las opciones que incluyen esta decisión. Avanzamos en la recursión para asignar al siguiente maestro, y posteriormente quitamos la partición creada para explorar otras opciones. Finalmente, devolvemos los valores calculados tras haber explorado todas las opciones.

```
1   def group_rec(benders_skills, n, index, partial_result, groups_result,
     min_coefficient,max_group_sum):
2       if has_a_greater_group_sum(partial_result, max_group_sum):
3           return min_coefficient, max_group_sum
4
5       coefficient = get_coefficient(partial_result)
6       if len(partial_result) == n and index == len(benders_skills) and
     coefficient <= min_coefficient:
7           max_group_sum = get_max_group_sum(partial_result)
8           for i in range(len(partial_result)):
9               groups_result[i].clear()
10              groups_result[i].extend(partial_result[i])
11          return coefficient, max_group_sum
12
13      if index == len(benders_skills) or coefficient > min_coefficient:
14          return min_coefficient, max_group_sum
15
16      for i in range(len(partial_result)):
17          partial_result[i].append(benders_skills[index])
18          min_coefficient, max_group_sum = group_rec(benders_skills, n, index +
     1, partial_result, groups_result, min_coefficient, max_group_sum)
19          partial_result[i].pop()
20
21      if len(partial_result) < n:
22          partial_result.append([benders_skills[index]])
23          min_coefficient, max_group_sum = group_rec(benders_skills, n, index +
     1, partial_result, groups_result, min_coefficient, max_group_sum)
24          partial_result.pop()
25      return min_coefficient, max_group_sum
```

4.4. Mención de “optimización” no incluida

Durante el análisis de las posibles optimizaciones que podríamos llevar a cabo, con la finalidad de que el agoritmo encuentre la solución óptima lo más pronto posible que significaría agilizar el calibramiento del pseudo-óptimo (en vista de lo conveniente que resulta como ya lo expusimos) para aprovechar de mejor forma las podas y optimizaciones que se basan en este, surgió la noción de que sería útil que para cada maestro el orden en que se prueben sus opciones (pertenecer al i-avo grupo) sea tal que primero se intente asignar al grupo que tenga una menor fuerza conjunta. Sin embargo, el implementar esta optimización resultó complejizando casi innecesariamente (sin una retribución equivalente) el algoritmo e incluso empeorando la complejidad espacial, pues, a medida que vamos profundizando en una rama el orden de estos grupos iría variando, ya que la decision que se toma en el nivel de profundidad anterior al actual potencialmete cambia cuál es el grupo con menor fuerza conjunta, por lo que tendríamos que ir volviendo a: calcular el el valor de la fuerza conjunta por cada grupo, copiar los grupos para no alterar el orden en que continuaría explorando el nivel anterior al hacer backtrack, y realizar un ordenamiento según las fuerzas conjuntas de estos. Además, más allá de las complicaciones respecto al espacio (copiar los grupos), tiempo (ordenar potencialmente 2^n veces un arreglo de K elementos) y simplicidad del código, al realizar pruebas iniciales con y sin esta implementación, empíricamete no percibimos una mejora significativa en los tiempos de ejecución. Es por tanto que, aunque teóricamente parecía una buena idea la optimización, finalemente esta no se incluyó en nuestra propuesta de solución por backtracking.

5. Aproximación greedy

En esta sección ampliaremos un poco las características que queremos que un algoritmo de aproximación al problema dado cumpla. Contando con la propuesta de aproximación del maestro Pakku, a continuación profundizaremos en la lógica del algoritmo así como también expondremos nuestra implementación del mismo en código, la complejidad temporal y espacial del mismo, la optimalidad empírica que alcanza el mismo.

5.1. Algoritmo

Generar los k grupos vacíos. Ordenar de mayor a menor a los maestros en función de su habilidad. Agregar al más habilidoso al grupo con menos habilidad conjunta hasta el momento. Repetir siguiendo con el siguiente más habilidoso, hasta que no queden más maestros por asignar.

5.2. Código

A continuación, presentamos nuestra propuesta de implementación de la aproximación referida por la cátedra, seguida de una breve explicación de cómo es que reflejamos lo indicado por el maestro Pakku, continuando con el análisis de su complejidad y culminando con una serie de comparaciones subyacentes a su naturaleza de aproximación (tiempos de ejecución en contraste con algoritmo exacto y optimaldidad en contraste con la solución óptima).

```
    def pakku_approach(benders_skills,k):
        groups = [[] for _ in range(k)]
        benders_skills = sorted(benders_skills,key=lambda x: x[1],reverse=True)
        i_most_skilled = 0
        groups_skills = [0]*k
        for bender,skill in benders_skills:
            i_weaker = groups_skills.index(min(groups_skills))
            groups[i_weaker].append((bender,skill))
            groups_skills[i_weaker] += skill
            if groups_skills[i_weaker]>groups_skills[i_most_skilled]:
                i_most_skilled = i_weaker
        coeficient = sum(s**2 for s in groups_skills)
        return groups, coeficient, groups_skills[i_most_skilled]
```

Nota: respecto a la implementación presentada en el código fuente, nuestro algoritmo de aproximación tambien devuelve los recursos que consideramos nos ahorran recálculos, con el fin de analizar la aproximación y no el programa en sí (que efectivamente se refleja en la sección de mediciones), inlcuimos en el presente una versión con la breve modiifcación mencionada.

5.3. Complejidad

Complejidad temporal: A continuación, presentaremos un breve desgloce de la complejidad temporal de los paso seguidos en el algoritmo proporcionado por la cátedra, el siguiente análisis denota a n como la cantidad total de maestros y k como la cantidad total de subgrupos de maestros (siguiendo la notación empleada en el resto del informe).

- Generación de los k grupos vacios: $O(k)$. Debido a que la inicialización de cada lista se realiza en tiempo constante y este paso se realiza k veces.
- Ordenar de mayor a menor a los maestros: $O(n * log(n))$. Esto debido a la complejidad de la función *"sorted"* ofrecida por el lenguaje (asumiendo se implementa un algoritmo de ordenamiento eficiente)
- Generación de la lista con las habilidades de cada grupo: $O(k)$. Debido al costo de inicializar cada posición de una lista con k ceros.

- Dentro del ciclo for que recorre todos los maestros (se repite n veces):
  - Obtener el índice del grupo con menos habilidad hasta el momento: $O(k)$ debido a que buscamos el índice del mínimo elemento en una lista no necesariamente ordenada y hay que recorrer todos los elementos.
  - Agregar al más habilidoso al grupo con menos habilidad hasta ahora: $O(1)$ debido a que en la implementación ya tenemos hallado tanto el índice del grupo más débil como el maestro más habilidoso (debido a que la lista está ordena por habilidad descendiente).
  - Actualizar la habilidad del grupo al que agregamos un maestro de la lista de habilidades totales por grupo: $O(1)$. Debido a que indexar en una lista se realiza en tiempo contante, de la misma forma que lo es: sumar un un número y volver a actualizar dicho elemento.
  - Actualizar el índice del grupo que acumula más habilidad hasta el momento: $O(1)$. Puesto que comparar dos valores se realiza en tiempo constante al igual que actualizar dicho valor.
- Calcular el coeficiente calculado en la partición ofrecida: $O(k)$. Debido a que le realizamos operaciones aritméticas a cada valor de una lista de k elementos, para luego sumar estos valores.

Es decir, la complejidad temporal del algoritmo de aproximación de Pakku es:

$$O(k) + O(n \times log(n)) + O(k) + n \times (O(k) + 3 \times O(1)) = O(n \times (log(n) + k))$$

Complejidad espacial: El siguiente análisis lo iniciaremos identificando las entidades de nuestro algoritmo que *viven en memoria* durante la ejecución de este (además de los datos de entrada):

- Grupos de maestros (solución aproximada): $O(n)$ puesto que la sumatoria de la cantidad de elementos de cada lista que contendrá a los subgrupos, en el peor de los casos será una particón de n elementos (maestros). Si se quiere se riguroso, en realidad guardamos tuplas para representar la información de cada maestro, con la notación *Bih-Oh!* esta constante desaparece.
- Total de maestros (dato de entrada y ordenamiento de este): $O(n)$. Puesto que la cantidad total de maestros es siempre n, y que la función de ordenamiento *sorted* proporcionada por el lenguaje tiene la complejidad espacial ya mencionada.
- Índice del grupo con menos habilidad conjunta, y coeficiente final calculado por nuestro algoritmo de aproximación: $O(1)$. Puesto que ambos son números.
- Habilidades conjuntas por cada grupo: $O(k)$. Puesto que tenemos k números en una lista (que se vam actualizando)
- Dentro del ciclo for que se repite n veces lo único que por cada iteración ocupa más espacio en memoria es la agregación del maestro más habilidoso a la solución aproximada (punto que ya se consideró previamente). El resto de variables que se crearon dentro de este ciclo ocupan espacio constante (como lo pueden ser variables auxiliares que almacenan indices de cálculos realizados que se actualizan con cada iteración).

Es decir, la complejidad espacial del algoritmo de aproximación de Pakku es:

$$O(k) + O(n) = O(k + n)$$

(Lineal respecto a la cantidad de particiones y la cantidad de maestros)

5.4. Optimalidad empírica como aproximador

Esta sección apunta al análisis de cuánto se pueden llegar a alejar los resultados obtenidos por el aproximador proporcionado de la solución óptima. Para ello, decidimos realizar una serie de mediciones que vendrán a mostrarnos de forma empírica como es que la solución obtenida por el presente approach no se aleja tanto de la verdadera solución.

5.4.1. Respecto a las prubeas ofrecidas por la cátedra:

A continuación observaremos cómo se comporta (en términos de eficacia) la estimación calculada respecto a instancias del problema de la tribu del agua con una cantidad reducida de opciones. Los datos que configuran cada instancia del P.T.A a los que haremos referencia a continuación son exactamente los que se pueden encontrar en el drive de la materia (link inlcuido en el apéndice)

1. Para una entrada con 5 maestros y $k = 2$.

Coeficiente óptimo	Coeficiente aproximado	Error absoluto	Porcentaje de error (%)
1894340	1894340	0	0

2. Para una entrada con 6 maestros.

	Coeficiente óptimo	Coeficiente aproximado	Error absoluto	Porcentaje de error (%)
$k = 3$	1640690	1640690	0	0
$k = 4$	807418	807418	0	0

3. Para una entrada con 8 maestros y $k = 3$.

Coeficiente óptimo	Coeficiente aproximado	Error absoluto	Porcentaje de error (%)
4298131	4298131	0	0

4. Para una entrada con 10 maestros.

	Coeficiente óptimo	Coeficiente aproximado	Error absoluto	Porcentaje de error (%)
$k = 3$	385249	385249	0	0
$k = 5$	355882	355882	0	0
$k = 10$	172295	172295	0	0

5. Para una entrada con 11 maestros y $k = 5$.

Coeficiente óptimo	Coeficiente aproximado	Error absoluto	Porcentaje de error (%)
2906564	2906564	0	0

6. Para una entrada con 14 maestros.

	Coeficiente óptimo	Coeficiente aproximado	Error absoluto	Porcentaje de error (%)
$k = 3$	15659106	15664276	5170	0.03302
$k = 4$	15292055	15292085	30	0.00020
$k = 6$	10694510	10700172	5662	0.05294

7. Para una entrada con 15 maestros.

	Coeficiente óptimo	Coeficiente aproximado	Error absoluto	Porcentaje de error (%)
$k = 4$	4311889	4317075	5186	0.12027
$k = 6$	6377225	6377501	276	0.00433

8. Para una entrada con 17 maestros.

	Coeficiente óptimo	Coeficiente aproximado	Error absoluto	Porcentaje de error (%)
$k = 5$	15974095	15975947	1852	0.01159
$k = 7$	11513230	11513230	0	0
$k = 10$	5427764	5430512	2748	0.05063

9. Para una entrada con 18 maestros.

	Coeficiente óptimo	**Coeficiente aproximado**	**Error absoluto**	**Porcentaje de error (%)**
$k = 6$	10322822	10325588	2766	0.02679
$k = 8$	11971097	12000279	29182	0.24377

10. Para una entrada con 20 maestros.

	Coeficiente óptimo	**Coeficiente aproximado**	**Error absoluto**	**Porcentaje de error (%)**
$k = 4$	21081875	21083935	2060	0.00977
$k = 5$	16828799	16838539	9740	0.05788
$k = 8$	11417428	11423826	6398	0.05604

De las mediciones previas, podemos observar que la diferencia (error absoluto) entre el coeficiente óptimo y el coeficiente aproximado, inicialmente (para espacios de soluciones reducidos) es nula, y a medida que agrandamos el universo de posibilidades los datos de la aproximación empiezan a alejarse más del valor esperado (la diferencia entre la aproximación y la solución óptima se hace tangible). Es más, observamos que para las muestras de datos procesadas:

- El **mayor** margen de error (procentaje de error) es **0.24377 %** y el **menor** (en el mejor de los casos) es **0 %** (La proximación es la solución óptima).
- La variabilidad del margen de error no depenede unicamente de k o de n.Es decir:
  - Para un mismo n observamos que hay aproximaciones que se acercan de manera diferente a la solución óptima.
  - Incrementar n no implica para todo caso que el margen de error incrementará (véase los resultados para $k = 6$, comparando para $n = 14$ y $n = 7$)
  - Incrementar k para un mismo n no necesariamente implica que incrementará el porcentaje de error (véase los resultados para una entrada de $n = 17$, comparando $k = 7$ y $k = 10$).

5.4.2. Respecto a nuestros sets de datos

Este apartado se abocará al análisis para sets de datos más grandes y variado, por lo mismo, la técnica que usamo para analizar los resultados obtenidos es diferente a la usada previamente:

- **Generación de sets de datos**: para mayor facilidad, fijaremos un valor de "k"para el cual produciremos diferentes conjuntos totales de maestros con diferentes valores de "n.
- **Procesamiento de los sets generados:** Calculamos el coeficiente aproximado y el coeficiente óptimo para cada instancia generada del problema. Procedemos a comparar los valores obtenidos
- **Sobre las gráficas:** Dados los resultados obtenidos, por cada conjunto de mediciones presentaremos 3 gráficas: las dos primeras que representan los coeficientes obtenidos en los algoritmos ejecutados (la aproximación y el valor óptimo) y el tercero que representa el margen de error de dichas mediciones (las 3 en función de n: cantidad de maestros). **Aclaración:** la superposición de las dos primeras gráficas teoricamente parece una buena forma de comparar los datos obtenidos; al realizar los respectivos test nos percatamos que al ser los coeficientes tan parecidos, las gráficas casi se superponian. Nos pareció más conveniente mostrar ambos resultados por separado usando y adicionar la tercera gráfica para facilitar la busqueda de la cota empírica.

.

1. **Resultados para** $k = 3$ $n \in [4; 25]$

Figura 1: Medición 1- Coeficiente aproximado

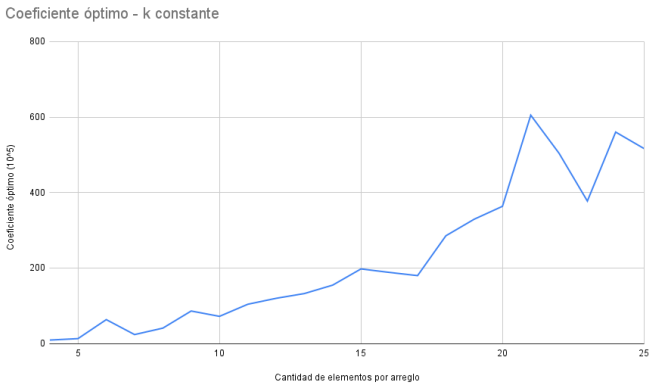

Figura 2: Medición 1 - Coeficiente óptimo

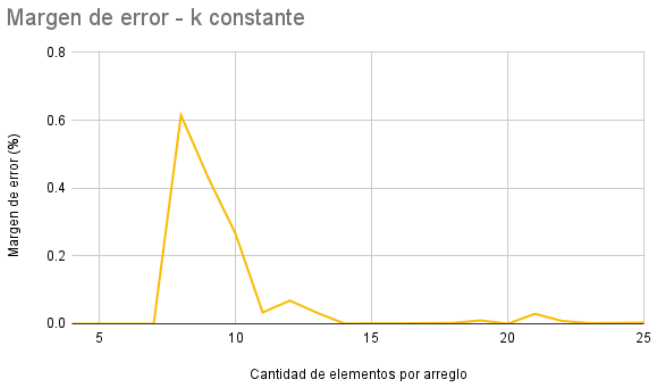

Figura 3: Medición 1 - Margen de error

2. **Resultados para** $k = 3$ $n \in [4; 25]$

Figura 4: Medición 2 - Coeficiente aproximado

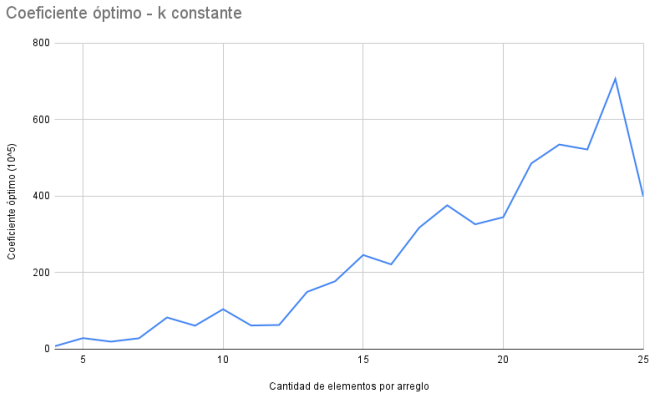

Figura 5: Medición 2 - Coeficiente óptimo

Figura 6: Medición 2 - Margen de error

3. **Resultados para** $k = 5$ $n \in [6; 22]$

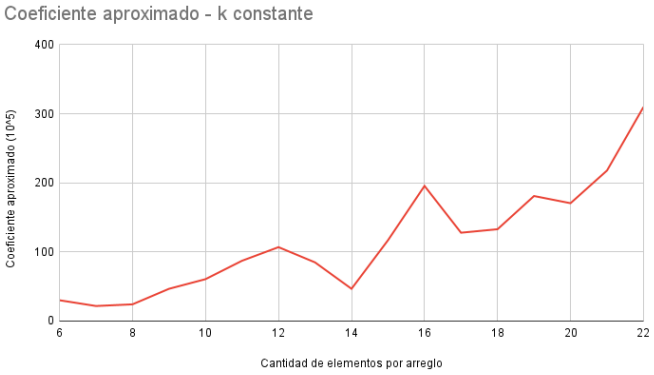

Figura 7: Medición 3 - Coeficiente aproximado

Figura 8: Medición 3 - Coeficiente óptimo

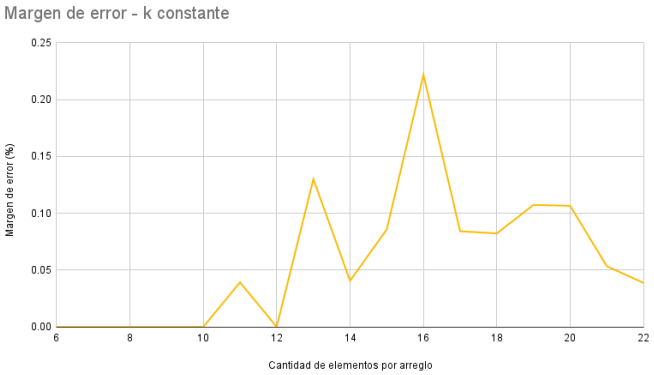

Figura 9: Medición 3 - Margen de error

4. **Resultados para** $k = 4$ $n \in [5; 25]$

Figura 10: Medición 4 - Coeficiente aproximado

Figura 11: Medición 4 - Coeficiente óptimo

Figura 12: Medición 4 - Margen de error

Podemos observar que para cada una de las mediciones, si bien los coeficientes obtenidos no son siempre los óptimos, no distan tanto del valor que se esperaría obtener. De hecho, es claro que para las mediciones tomadas: en el peor de los casos el margen de error de nuestra aproximación es casi del 0.6 % (un poco más). Debido a que los resultados no son concluyentes pues son una serie finita de mediciones que podrían excluir algún caso en el que se encuentre otro margen de error máximo que supere a los encontrados hasta las mediciones previas, proponemos que el margen de error de

la aproximación usada es del 1 % (para abarcar posibles errores no encontrados en las mediciones tomadas) aun que en los experimentos realizados no encontramos aprximaciones con un margen de error superior al 0.7 %. Podemos concluir que la presente aproximación resulta cercana al valor real esperado en la mayoria de los casos.

6. Construcción de la solución por programación lineal

Variables del modelo:

Para encarar el problema desde un enfoque de programación lineal plantearemos el problema a partir de variables binarias que representen la inclusión o no de un maestro i en el grupo j, siendo que $S_{i,j} = 1$ representa la inclusión del maestro i en el grupo j y $S_{i,j} = 0$ la no inclusión del mismo en el grupo j. Por lo tanto, el primer paso en nuestro algoritmo resulta siendo la inicialización de estas variables, que en total resultan siendo $n \times k$, siendo n la cantidad de maestros en total y k la cantidad de grupos que se desea armar. La inicialización se implementó en código de la siguiente forma:

```
    def lp_algorithm(benders_skills, groups_count):
        S = []
        benders_count = len(benders_skills)

        for i in range(groups_count):
            S.append([])
            for k in range(benders_count):
                S[i].append(pulp.LpVariable("S_"+str(i)+","+str(k),cat="Binary"))
```

Siendo S la matriz de nxk mediante la cual accederemos a las variables binarias que representan la inclusion de un maestro en un grupo.

Definición del Problema de Optimización:
Una vez inicializadas las variables binarias con las que trabajamos definimos el problema de optimización, en nuestro caso el problema consiste en minimizar la función objetivo que posteriormente dearrollaremos cómo una expresarla como una ecuación lineal *(Véase: Linealización del cuadrado de la sumatoria de un grupo)* La definición del problema se implementa en código de la siguiente forma:

```
        problem = pulp.LpProblem("benders", pulp.LpMinimize)
```

Restricciones del problema:
A continuación nos encargamos de modelar ecuaciones lineales tales que logren reflejar las limitaciones de la inclusión de un maestro en un grupo o no, estas serían que para cada maestro (n) se debe cumplir que pertenezca exactamente a uno de los k grupos.

Para plasmar las limitaciones mencionadas en ecuaciones lineales que se puedan incluir en el modelo lineal, propusimos que:

- Para $j_0 = 0, 1, ..., n-1$, se establece la siguiente ecuación: $\sum_{i=0}^{k-1} S_{i,j_0} = 1$, lo que quiere denotar que las variables que representan la inclusión de un mismo maestro j_0 en los diferentes i grupos sea exactamente 1.

Lo mencionado se implementó en código de la siguiente forma:

```
    for k in range(benders_count):#n
        problem += pulp.lpSum([S[i][k] for i in range(groups_count)]) == 1
```

Ahora bien, la principal problemática de resolver el problema de la tribu del agua se presenta en cómo **describir la función objetivo del P.T.A de forma lineal** siendo que la misma es:

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

una función cuadrática. A continuación expondremos cómo encaramos esta problemática aprovechando la naturaleza binaria (que puede verse como booleana) de las variables que ya definimos en nuestro modelo.

6.1. Linealización del cuadrado de la sumatoria de un grupo

Para evadir operaciones cuadráticas en nuestras ecuaciones resulta necesario recordar la naturaleza binaria de nuestras variables, ya que para poder encarar el problema mediante programación lineal resulta necesario el poder linealizar nuestra función objetivo (valga la redundancia).

> ***Observación 5.-*** La multiplicación de dos variables binarias se puede interpretar como una operación *and* entre dos variables booleanas, por lo que si definimos una nueva variable que represente que ambas variables valgan 1 podemos deshacernos de las ultiplicaciones entre variables binarias (operaciones no lineales)

Partamos linealizando una suma de cuadrados ordinaria:

$$(a_1x_1 + a_2x_2 + ... + a_ix_i)^2 = (a_1x_1 + a_2x_2 + ... + a_ix_i)(a_1x_1 + a_2x_2 + ... + a_ix_i) =$$
$$[(a_1a_1x_1x_1) + (a_1a_2x_1x_2) + ... + (a_1a_ix_1x_i)] + ... + [(a_ia_1x_ix_1) + (a_ia_2x_ix_2) + ... + (a_ia_ix_ix_i)]$$

Y por lo visto en la *Obseración 5* podemos reescribir x_ix_j como: $And(x_i, x_j)$ (recordemos también que $And(x_0, x_0) = x_0$) entonces:

$$[(a_1a_1x_1) + (a_1a_2And(x_1, x_2)) + ... + (a_1a_iAnd(x_1, x_i))] + ... +$$
$$[(a_ia_1And(x_i, x_1)) + (a_ia_2And(x_i, x_2)) + ... + (a_ia_ix_i)]$$

Con lo que, definiendo variables auxiliares que representen $And(x_i, x_j)$ entre dos variables binarias podemos lograr linealizar la expresión de la suma de cuadrados, misma que es la función objetivo en el problema de la tribu del agua.

> ***Nótese*** que si bien podemos ver multiplicaciones entre lo que a primera vista podrían parecer variables: a_ia_j, realmente estas son constantes, las unicas variables en la ecuación son x_i y $And(x_i, x_j)$, mismas que en la expresión a la que llegamos solo están multiplicadas por escalares constantes

6.2. Definicion de variables auxiliares

Entonces, como ya establecimos previamente, resulta necesario agregar a nuestro modelo variables auxiliares (también binarias) que representen que entre dos variables ya existentes en nuestro modelo ambas valen 1.

La inicialización de estas, así como la restriccion para que cada una efectivamente tenga la funcionalidad del operador *And* la implementamos en código de la siguiente forma:

```
auxiliar_and_vars = []
for i in range(groups_count):#k
    auxiliar_and_vars.append([])
    for j in range(benders_count):#n
        auxiliar_and_vars[i].append([])
        n = 2 # And de dos binarios
        for k in range(benders_count):
            if k == j:
                #Caso and de una misma variable
                auxiliar_and_vars[i][j].append(S[i][j])
                continue
            if k < j:
                # Caso de un and repetido: x1 and x2=x2 and x1
                auxiliar_and_vars[i][j].append(auxiliar_and_vars[i][k][j])
                continue
            auxiliar_and_vars[i][j].append(pulp.LpVariable(f"Group_{i}_and_of_[{j
    }-{k}]", cat="Binary"))

            #restricciones para que ser efectivamente and's en el modelo
            problem+=n*auxiliar_and_vars[i][j][k]<=pulp.lpSum([S[i][j],S[i][k]])
            problem += (n - 1) + auxiliar_and_vars[i][j][k] >= pulp.lpSum([S[i][j],
     S[i][k]])
```

6.3. Expresion final: función objetivo

Ahora bien, respecto a la función objetivo, esta misma ya se puede explicitar como una expresión lineal en función a las variables de inclusión $S_{i,j}$, las auxiliares recién expuestas y de coeficientes a_i los valores de las fuerzas respectivas al maestro j. Con lo cual algorítmicamente podemos expresar la función objetivo de la siguiente forma:

```
problem += pulp.lpSum([pulp.lpSum([(auxiliar_and_vars[i][j][k]*(benders_skills[j
    ][1]*benders_skills[k][1])) for j in range(benders_count) for k in range(
    benders_count)]) for i in range(groups_count)])
```

> ***Recordar*** que el segundo elemento de la $n-ava$ tupla en *benders_skills* es la fuerza del n-avo maestro.

6.4. Resolución del Problema y reconstrucción de la solución

Una vez incluidas las restricciones expuestas a lo largo del desarrollo de la presente sección en el modelo de problema lineal, finalmente podemos pedirle al solver de pulp que encuentre aquellos valores de las variables definidas, tales que minimicen el valor de la función objetivo. Esto lo logramos mediante la instrucción:

```
    problem.solve(pulp.PULP_CBC_CMD(msg=0))
```

Una vez hecho esto podemos extraer los valores resultantes que requiramos para llevar a cabo la reconstrucción de los grupos que se formaron con la inclusión de los n maestros en alguno de estos. Para esto nos manejamos con la matriz S en la que organizamos las variables de nuestro interés: $S_{i,j}$ la inclusion o no en el grupo i del maestro j.

Entonces recopilando y procesando los valores óptimos de las variables binarias podemos llegar a la lista de grupos de maestros que representa una solución óptima al P.T.A. Este proceso se implementó algorítmicamente de la siguiente manera:

```
    groups = []
    for group_binary_values in binary_results:
        group = []
        for i in range(len(group_binary_values)):
            if group_binary_values[i] == 1:
                group.append(benders_skills[i][0])
        groups.append(group)
```

Además, en vista de que pudimos linealizar la misma función objetivo del problema original, disponemos del valor esta, pues el solver de pulp es quien lo calculó, entonces, mediante la siguiente instrucción lo obtenemos:

```
    coefficient = int(problem.objective.value())

```

Con lo que, finalmente, mediante el algoritmo descrito, pudimos obtener la distribución de los n maestros en k grupos (*groups*) que repsente una solcuión óptima. De igual forma también disponemos del valor de la función objetivo minimizado.

6.5. Comlejidad temporal

Respecto a lo que es la complejidad temporal teórica de resolver un problema de programación lineal entera utilizando PuLP y su solver CBC esta es típicamente exponencial en función del número de variables, con el número de restricciones también influyendo en la complejidad. Cabe mencionar que, si bien la complejidad temporal de un problema de programación lineal entera se suele evaluar empíricamente, esta resultó una tarea intrincada, pues los tiempos de ejecución del algoritmo para distintos sets de pruebas se iban haciendo muy extensos.

También resaltamos que esta fue la heurística de entre las implementadas que más tardó. Razón por la cual en las mediciones de tiempos de ejecución de este algoritmo tuvimos que limitar considerablemente la variedad de estas.

7. Mediciones

7.0.1. Procesamiento y generación de datos de entrada:

Se realizaron mediciones en base a crear arreglos de hasta un largo n, para los cuales, en cada caso, los elementos fueron generados por los valores pseudoaleatorios del lenguaje (el módulo `random`).

Dividimos las mediciones en n-constante y k-constante, siendo n la cantidad de maestros y k la cantidad de grupos

7.1. Medición - Algoritmo Backtracking

Medición n-constante:

Una vez generado el arreglo de n elementos, se ejecutó el algoritmo incrementando k en 1 por cada llamado hasta alcanzar n

Figura 13: Medición 1

Medición k-constante:

Con los arreglos generados, se fijó una cantidad de grupos k para la ejecución de cada uno hasta el arreglo de largo n

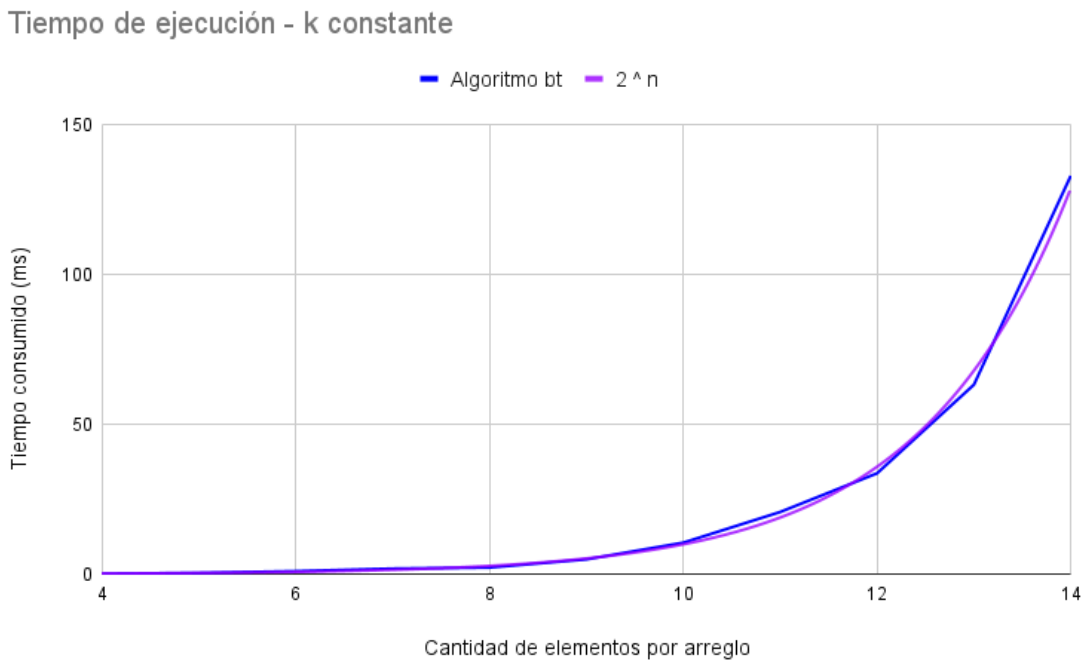

Figura 14: Medición 1

7.1.1. Análisis de los resultados:

Con estos resultados es preciso el poder afirmar que el algoritmo por backtracking propuesto, efectivamente tiene una tendencia **exponencial** en función del tamaño de los datos de entrada, debido a la naturaleza de backtracking.

Además podemos notar que los tiempos del caso $n - constante$ son significativamente mayores al de $k - constante$, esto nos muestra que la cantidad de grupos k tiene una gran influencia en los tiempos del algoritmo.

7.2. Medición - Algoritmo de aproximación Greedy

Medición n-constante:

Una vez generado el arreglo de n elementos, se ejecutó el algoritmo incrementando k en 1 por cada llamado hasta alcanzar n

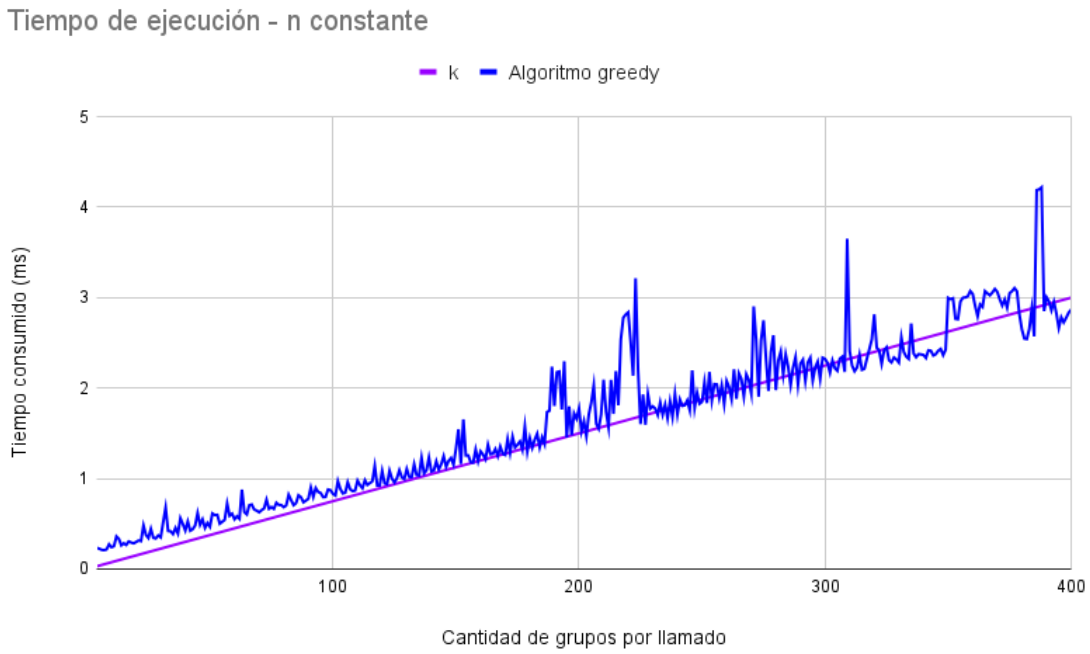

Figura 15: Medición 1

Medición k-constante:

Con los arreglos generados, se fijó una cantidad de grupos k para la ejecución de cada uno hasta el arreglo de largo n

Figura 16: Medición 1

7.2.1. Análisis de los resultados:

Podemos observar que la tendencia del caso $n-constante$ es **lineal** mientras que en $k-constante$ resulta **n-logaritmica**.

Estos resultados son debidos al ordenamiento inicial de los n elementos $O(n.logn)$, que se ven opacados por la complejidad del algoritmo $O(n.k)$.

7.3. Medición - Algoritmo Greedytracking

Medición n-constante:

Una vez generado el arreglo de n elementos, se ejecutó el algoritmo incrementando k en 1 por cada llamado hasta alcanzar n

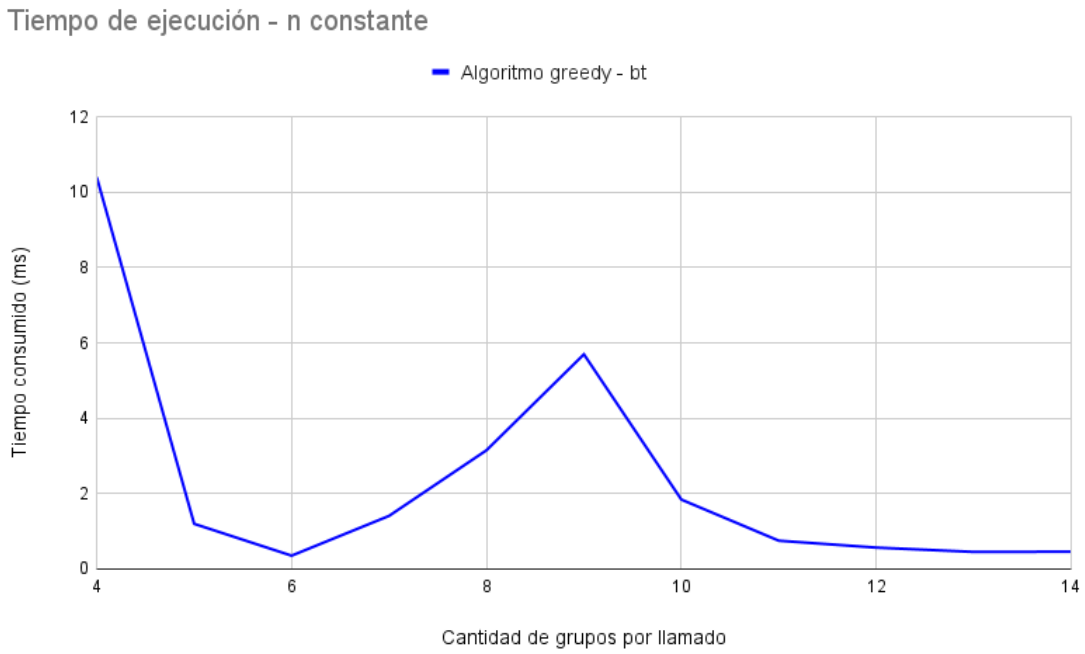

Figura 17: Medición 1

Medición k-constante:

Con los arreglos generados, se fijó una cantidad de grupos k para la ejecución de cada uno hasta el arreglo de largo n

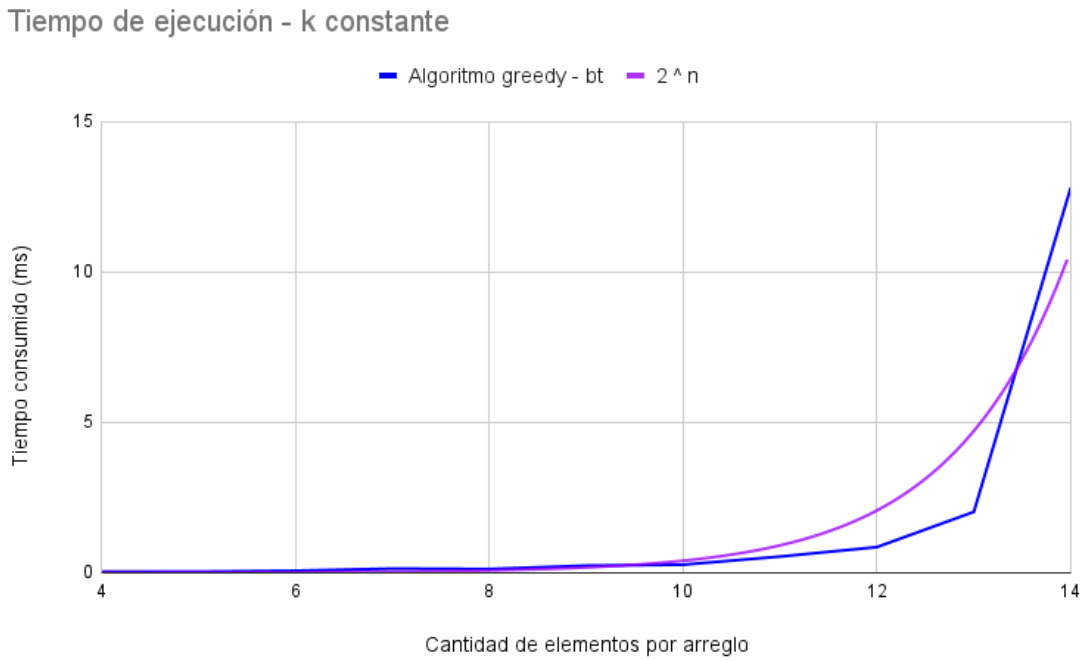

Figura 18: Medición 1

7.3.1. Análisis de los resultados:

Podemos observar que en el $n - constante$ el tiempo de ejecución se reduce conforme k se acerca a n, esto debido a las podas realizadas respecto a la aproximación greedy que además obtienen una mejoría muy significativa en los tiempos de ejecución del caso $k - constante$.

7.4. Medición - Algoritmo PL

Medición n-constante:

Una vez generado el arreglo de n elementos, se ejecutó el algoritmo incrementando k en 1 por cada llamado hasta alcanzar n

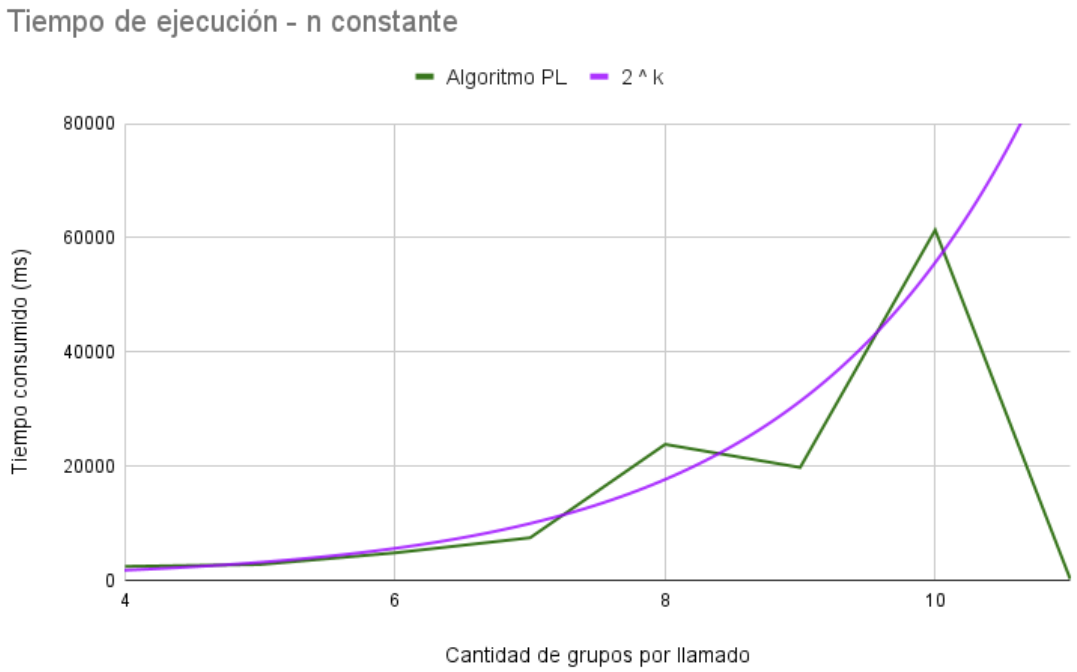

Figura 19: Medición 1

Medición k-constante:

Con los arreglos generados, se fijó una cantidad de grupos k para la ejecución de cada uno hasta el arreglo de largo n

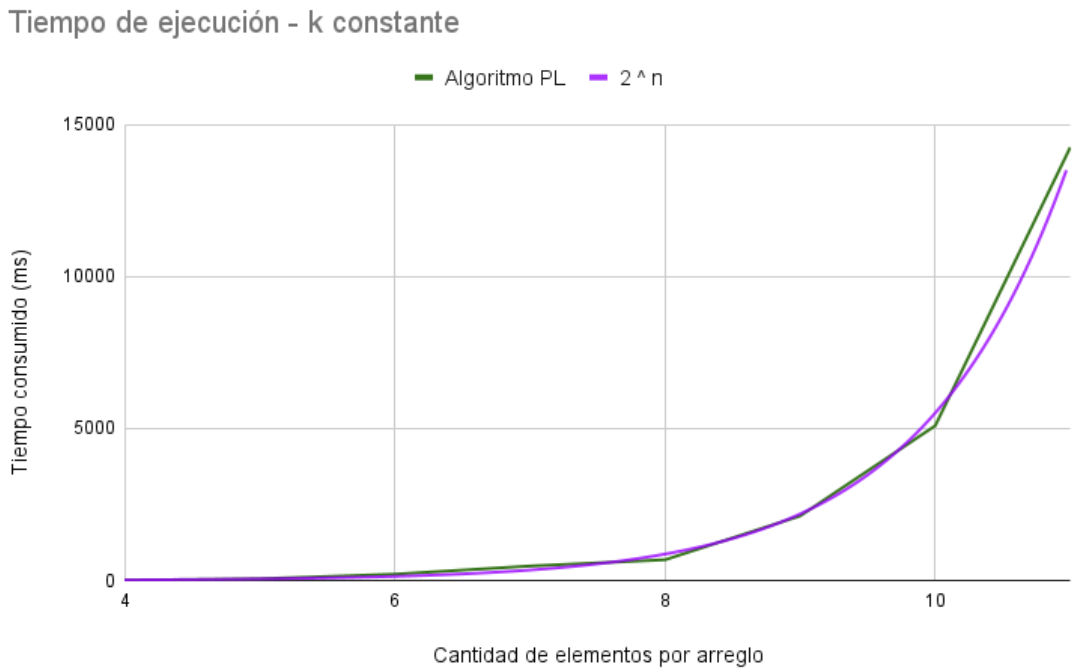

Figura 20: Medición 1

7.4.1. Análisis de los resultados:

Podemos observar que tanto la tendencia del caso $n - constante$ como $k - constante$ resulta **exponencial**. Esto es de esperar debido a la naturaleza e implementación de la programación lineal entera.

8. Conclusiones

En el presente trabajo se abordaron distintas estrategias para solucionar el problema de la distribución óptima de los maestros de la Tribu del Agua con el objetivo de mantener un ataque constante y eficiente contra la Nación del Fuego. A continuación, se resumen las conclusiones más relevantes derivadas del análisis y la implementación de estas estrategias:

- Respecto al **análisis de la complejidad del problema:** Se confirmó que el problema pertenece a la clase NP-Completo, lo cual implica que no existe un algoritmo eficiente conocido para resolverlo en todos los casos posibles dentro de un tiempo razonable. La clasificación del problema se logró a través de una estrategia de reducción que lo relaciona con el problema NP-Completo de 2-partition.

Las **implementaciones de soluciones** para encarar el problema mediante distintas técnicas de diseño nos permitieron llegar a las siguientes conluciones:

- Respecto a la estrategia implementada mediante **backtracking**: usando esta técnica de diseño de solución se generan y exploran todas las posibles combinaciones de asignación de maestros a subgrupos a priori, con lo cual nos cercioramos de obtener siempre la solución óptima, pues la eurística de esta técnica es **exhaustiva**. Dado que esta exploración puede resultar muy costosa respecto a los recursos temporales resulta importante incluir **podas y optimizaciones** para mejorar el rendimiento del mismo, entre las cuales destaca la consideración de una primera aproximación greedy como estándar mínimo para las siguientes exploraciones, dicha optimización permitió descartar rápidamente soluciones no prometedoras, puesto que la aproximación greedy resulta, empíricamente hablando, una muy buena aproximación.
- Respecto a la **solución por programación lineal:** Se formuló una aproximación de programación lineal que incluyó la linealización de la sumatoria cuadrática de las fuerzas de los maestros en cada grupo y la definición de variables auxiliares. La función objetivo y las restricciones se diseñaron para garantizar una distribución equitativa y eficiente de los maestros. Cabe resaltar que, no resultó una tarea sencilla el plantear las ecuaciones lineales y manejar una cantidad de variables en el modelo tan extensa, además, grupalmente no percibimos que esta eurísitca haya resultado retribuyente pues, en cuanto a tiempos de ejecución esta fue la técnica que requirió de más tiempo de espera, razón por la cual también se volvieron tareas especialmente laboriosas la realización de pruebas y el análisis del resultado de las mismas.
- Respecto a la aproximación **greedy**, al implementar un algoritmo de esta heurística pudimos demostrar que la misma resulta efectiva para generar soluciones aproximadas en tiempos reducidos (polinómicos) que resulta miuy útil en el contexto de un problema NP-Competo, es decir, del que esperamos un costo considerablemente alto para encontrar la solución a este.

  La eficacia empírica de esta aproximación se evaluó con diferentes conjuntos de datos, mostrando un desempeño adecuado con errores mínimos respecto a la solución óptima en las pruebas realizadas.

Los **resultados empíricos** obtenidos mediante pruebas con conjuntos de datos proporcionados por la cátedra así como los generados grupalmente, demostraron que las soluciones obtenidas por el algoritmo greedy fueron en muchos casos óptimas o muy cercanas al óptimo, validando así la eficiencia de esta aproximación.

En resumen, el informe evidencia que, aunque el problema de la distribución de los maestros de la Tribu del Agua es intrínsecamente complejo por ser este un problema NP-Completo, las soluciones implementadas, especialmente la aproximación greedy, ofrecen un balance adecuado entre precisión y eficiencia. Las técnicas de programación lineal y algoritmos de búsqueda exhaustiva demuestran ser una estrategias robustas para enfrentar problemas de esta naturaleza.

Sin embargo, al realizar una **"colaboración" entre los resultados de las técnicas: Greedy y Backtracking** (incluyendo optimizaciones y podas), pudimos alcanzar un algoritmo, que en lo que respecta al análisis grupal, se ejecuta en un tiempo satisfactorio, además, es de nuestra consideración que el mismo garantiza obtener la solución óptima.