.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

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

13 de junio de 2024

Alan Ramiro Cueto Quinto
104319

Carlos Matias Sagastume
110530

Thiago Fernando Baez
110703

1. Introducción

En este trabajo práctico tenemos el objetivo de analizar el problema a continuación y evaluar y desarrollar un algoritmo por Backtracking que resuelva dicho problema.
El problema en cuestion es:
Es el año 95 DG. La Nación del Fuego sigue su ataque, esta vez hacia la Tribu del Agua, luego de una humillante derrota a manos del Reino de la Tierra, gracias a nuestra ayuda. La tribu debe defenderse del ataque.

El maestro Pakku ha recomendado hacer lo siguiente: Separar a todos los Maestros Agua en k grupos $(S_1, S_2, ..., S_k)$. Primero atacará el primer grupo. A medida que el primer grupo se vaya cansando entrará el segundo grupo. Luego entrará el tercero, y de esta manera se busca generar un ataque constante, que sumado a la ventaja del agua por sobre el fuego, buscará lograr la victoria.

En función de esto, lo más conveniente es que los grupos estén parejos para que, justamente, ese ataque se mantenga constante.

Conocemos la fuerza/maestría/habilidad de cada uno de los maestros agua, la cuál podemos cuantificar diciendo que para el $maestro_i$ ese valor es x_i, y tenemos todos los valores $x_1, x_2, ..., x_n$ (todos valores positivos).

Para que los grupos estén parejos, lo que buscaremos es minimizar la adición de los cuadrados de las sumas de las fuerzas de los grupos. Es decir: $min \sum_{i=1}^{k} (\sum_{x_j \in S_i} x_j)^2$

El Maestro Pakku nos dice que esta es una tarea difícil, pero que con tiempo y paciencia podemos obtener el resultado ideal.

Además se nos pide demostrar que la versión de decisión del problema es NP y además es NP-Completo, también escribir un modelo de programación lineal con dos alternativas para hacerlo, en nuestro caso elegimos la segunda alternativa es decir escribir un modelo de programación lineal cuyo objetivo sea minimizar la diferencia del grupo de mayor suma con el de menor suma. Y finalmente, implementar el algoritmo dado por el Maestro Pakku, analizar su complejidad y analizar cuan buena aproximación es. Por lo tanto lo primero que haremos será la parte de la demostración, luego desarrollaremos acerca del algoritmo de Backtracking, después haremos el modelo de programación lineal y finalmente implementaremos el algoritmo Greedy pedido.

2. Demostración

Lo primero que hay que hacer es demostrar que el problema esta en NP, para eso hay que construir un verificador que dada una solucion al problema verifique en tiempo polinomial si es correcto o no. Para esto cree un verificador que recibe a la totalidad de los maestros agua, los subconjuntos, el número k de subconjuntos y el número B que no debe exceder la suma de cuadrados. El codigo de este verificador es:

```
def verificador(elementos, solucion, k ,b):
    if len(solucion) > k:
        return False
    sumatoria = 0
    elementosCubiertos = set()
    elementos = set(elementos)
    for subconj in solucion:
        for elemento in elementos:
            if elemento not in elementos:
                return False
            if elemento in elementosCubiertos:
                return False
            elementosCubiertos.add(elemento)
        sumatoria += sum(subconj) ** 2
    return sumatoria <= b and len(elementosCubiertos) == len(elementos)
```

Este verificador es polinomial respecto a la cantidad de maestros agua ya que primero crea un set con todos los maestros lo cual es lineal respecto a la cantidad de maestros aguas, luego itera todos

los sets y cada elemento de los sets y como los sets son una particion de el array de elementos original, entonces iterar todos los elementos de todos los sets es lineal respecto a la entrada, luego calcula la sumatoria de cada set lo cual se puede acotar como O(n) y eleva esa sumatoria al cuadrado lo cual es O(1) y finalmente suma ese resultado a la sumatoria total, por lo tanto todo el algoritmo es O(n) siendo n la cantidad de maestros agua y de este modo queda demostrado que el problema de la Tribu del Agua pertenece a NP. Ahora queda demostrar que el problema es NP-Completo para lo cual utilizaremos el partition problem el cual es un problema NP-Completo cuyo problema de decisión es: "Dado un set de enteros positivos determinar si puede ser partido en dos Subsets S_1 y S_2 tal que la sumatoria de los elementos de S_1 es igual a la sumatoria de los elementos de S_2. Primero demostraremos en que NP-Completo, primero para demostrar que pertenece a NP es sencillo ya que si se nos da 2 sets que dicen cumplir el problema basta con hacer la sumatoria de los elementos de ambos sets y probar que sean iguales, esto es lineal respecto a la cantidad de elementos original. Y para probar que es NP-Completo se puede reducir Subset Sum a este problema, esto se logra agregando al set S original 2 elementos: $z_1 = sum(S)$ y $z_2 = 2T$ siendo T la suma pedida por el problema original. Entonces si llamamos M al nuevo subset: $sum(M) = 2sum(S) + 2T$ por lo tanto si le pasamos como parametro a partition set al conjunto M, si tiene solucion significa que existen 2 conjuntos S' y S" tales que la sumatoria de cada uno es $sum(s) + T$ y si S' es el conjunto que posee a z_1 entonces el conjunto $S' - z_1$ cumple con lo pedido por problema original y por lo tanto el problema queda resuelto en cambio si partition set da false significa que no existen los conjuntos S' y S" y por lo tanto, el problema original no tiene solución. Es de esta manera que Subset Sum queda reducido al Partition Problem y luego partition problem es NP-Completo. Por lo tanto si reducimos este problema al problema de la Tribu del Agua habremos demostrado que el último es NP-Completo. La demostración es así: Llamemos K a la sumatoria de los elementos de S_1, notemos que la sumatoria de ambos sets es $2K$. Ademas notemos que si dividimos el set original en dos subsets que no necesariamente tienen la misma sumatoria, a los cuales llamaremos M_1 y M_2 tal que:
$\sum_{x \in M_1} x_i = K + m$ y $\sum_{x \in M_2} x_i = K - m$
Entonces:
$\sum_{x \in M_1} x_i + \sum_{x \in M_2} x_j = K - m + K + m = 2K$
Y notemos que:
$(\sum_{x \in M_1} x_i)^2 + (\sum_{x \in M_2} x_j)^2 = 2K^2 + 2m^2$
Por lo tanto si m = 0 tenemos que M_1 y M_2 son nada mas y nada menos que S_1 y S_2 y que además tienen la menor de la sumatoria de cuadrados entre todas las particiones en 2 de el set original, la cual es $2K^2$. Por lo tanto si por cada elemento i del set creamos un maestro i cuya habilidad es el valor del elemento i entonces cada elemento del set quedará representado por un maestro agua y si usamos la caja negra que resuelve el problema de la Tribu del Agua y le pasamos como parametro el array de maestros que recien creamos, como k le pasamos 2 y como B le pasamos $2K^2$ entonces si el algoritmo devuelve true significa que existe una particion del set original en dos subconjuntos tales que la suma del cuadrado de la sumatoria de sus elementos es menor o igual a $2K^2$ y como ya vimos la suma del cuadrado de la sumatoria de S_1 y S_2 es la menor de todas y es $2K^2$ por lo tanto si da true significa que existen los subconjuntos S_1 y S_2 buscados por el problema original, por lo cual el partition problem tiene solución y de la misma manera si da false significa que el partition problem no tiene solución. Por lo tanto de esta manera hemos reducido al partition problem al problema de la Tribu del Agua y como ya vimos que este problema pertenece a NP entonces podemos concluir que el problema de la Tribu del Agua es NP-Completo.

Ejemplo de Resolución

3. Aproximación por algoritmo Greedy

Como ya hemos demostrado que el problema es NP-Completo ahora presentaremos y analizaremos la implementacion del algoritmo Greedy pedido en el punto 5 que encuentra una aproximación a la solución del problema, el código del algoritmo es el siguiente:

```
def greedy1(maestros, k):
    maestros = sorted(maestros, key=lambda x: -x[1])
    conjuntos = []
    for i in range(k):
        conjuntos.append(set())
    for elemento in maestros:
        minimo = encontrar_minimo(conjuntos)
        minimo.add(elemento)
    return conjuntos


def encontrar_minimo(conjuntos):
    minimo = conjuntos[0]
    for conj in conjuntos:
        if sumatoria(conj) < sumatoria(minimo):
            minimo = conj
    return minimo


def sumatoria(conj):
    res = 0
    for elemento in conj:
        res += elemento[1]
    return res


def calcular_coeficiente(grupos):
    res = 0
    for g in grupos:
        res += sumatoria(g) ** 2
    return res
```

Este algoritmo implementa lo pedido por la consigna y a continuación analizaremos su complejidad: Primero como se ordena a la lista de maestros recibida, la complejidad del algoritmo sera por lo menos de O(n log n) debido a que la funcion sorted utiliza el algoritmo de ordenamiento Timsort que tiene esa complejidad (siendo n el tamaño de la entrada) luego, crear los k conjuntos es O(k) pero como k es menor o igual a n entonces se puede acotar a O(n). Luego se realiza un ciclo que itera todos los elementos una vez y dentro del ciclo se utiliza la función encontrar mínimo que es O(n), ya que la función sumatoria para calcular la sumatoria de todos los conjuntos debe sumar los elementos de cada conjunto y eso en total para todos los conjuntos es O(n), luego como para cada elemento se llama a una función O(n) finalmente el ciclo es $O(n^2)$ y como el ciclo es lo ultimo que se realiza, entonces la complejidad final es $O(n) + O(nlog(n)) + O(n) + O(n^2) = O(n^2)$ por lo tanto la complejidad del algoritmo resulta ser $O(n^2)$. Más adelante se haran las comparaciones con el algoritmo de Backtracking ya que era necesario presentar este algoritmo antes que el primero debido a que este algoritmo se utiliza como aproximación para el algoritmo de Backtracking.

4. Algoritmo de Backtracking

El algoritmo que se presentará a continuación encuentra por medio de backtracking la solución óptima al problema, utilizando como solucion inicial la aproximación dada por el algoritmo greedy presentado anteriormente. El código del algoritmo en cuestión es el siguiente:

```
import greedy1

```

```
SALTO_DE_PAGINA = "\n"
SEPARADOR = ", "
POS_HABILIDAD = 1


def backtracking(archivo):
    k, maestros = leer_archivo(archivo)
    aproximacion = greedy1.greedy1(maestros, k)
    res = [aproximacion, calcular_sumatoria_grupo(aproximacion)]  # Empiezo con una aproximacion
    conjuntos = []
    for i in range(k):
        conjuntos.append(set())
    _backtracking(maestros, conjuntos, 0, res)
    return res[0]


def _backtracking(maestros, conjuntos, actual, resultado_actual):
    if es_mayor(conjuntos, resultado_actual[1]) or ya_no_llega(conjuntos, resultado_actual[1], actual, maestros):
        return
    if actual == len(maestros):
        nuevo = copiar_grupos(conjuntos)
        resultado_actual[0] = nuevo
        resultado_actual[1] = calcular_sumatoria_grupo(nuevo)
        return
    for conj in conjuntos:
        conj.add(maestros[actual])
        _backtracking(maestros, conjuntos, actual + 1, resultado_actual)
        conj.remove(maestros[actual])
        if not conj:
            break


def copiar_grupos(conjuntos):
    res = []
    for g in conjuntos:
        res.append(g.copy())
    return res


def es_mayor(nuevo, actual):
    sumatoria_nuevo = calcular_sumatoria_grupo(nuevo)
    return sumatoria_nuevo >= actual


def ya_no_llega(nuevo, actual, n, maestros):
    sumatoria_nuevo = calcular_sumatoria_grupo(nuevo)
    for i in range(n, len(maestros)):
        sumatoria_nuevo += maestros[i][POS_HABILIDAD] ** 2
    return sumatoria_nuevo >= actual


def calcular_sumatoria_grupo(grupo):
    res = 0
    for c in grupo:
        res += sumatoria(c) ** 2
    return res


def sumatoria(conj):
    res = 0
    for elemento in conj:
        res += elemento[POS_HABILIDAD]
    return res


def calcular_coeficiente(grupos):
    res = 0
    for g in grupos:
```

```
        res += sumatoria(g) ** 2
    return res


def leer_archivo(archivo):
    maestros = []
    with open(archivo) as arch:
        arch.readline()
        k = int(arch.readline().rstrip(SALTO_DE_PAGINA))
        for linea in arch:
            nombre, habilidad = linea.rstrip(SALTO_DE_PAGINA).split(SEPARADOR)
            maestros.append((nombre, int(habilidad)))
    return k, maestros
```

Y el ejecutable main corre el siguiente código:

```
import backtracking
import sys

UBICACION_RUTA = 1
UBICACION_COMANDO = 2
UBICACION_RUTA_GUARDAR = 3
GUARDAR = "-guardar"
TEXTO_GRUPO = "Grupo"
SEPARADOR_GRUPOS = ", "
SEPARADOR = ":"
SALTO_DE_LINEA = "\n"
TEXTO_COEFICIENTE = "Coeficiente: "
MENSAJE_ERROR_PARAMETROS = "Error: parametros insuficientes"
MENSAJE_ERROR_RUTA = "Error: Ruta Invalida"
RUTA_GENERICA = "resultado.txt"
POS_NOMBRE = 0


def main():
    args = sys.argv
    ruta = args[UBICACION_RUTA]
    if len(args) < 1:
        print(MENSAJE_ERROR_PARAMETROS)
        return
    rutaGuardado = None
    if len(args) > 2:
        if args[UBICACION_COMANDO] == GUARDAR:
            rutaGuardado = args[UBICACION_RUTA_GUARDAR]
    grupos = backtracking.backtracking(ruta)
    if not rutaGuardado:
        rutaGuardado = RUTA_GENERICA
    try:
        with open(rutaGuardado, "w") as archivo:
            i = 1
            for g in grupos:
                archivo.write(f"{TEXTO_GRUPO} {i}{SEPARADOR} {unir_grupo(g)}{
    SALTO_DE_LINEA}")
            archivo.write(f"{TEXTO_COEFICIENTE} {backtracking.calcular_coeficiente(
    grupos)}")
    except Exception:
        print(MENSAJE_ERROR_RUTA)


def unir_grupo(grupo):
    res = []
    for elem in grupo:
        res.append(elem[POS_NOMBRE])
    return SEPARADOR_GRUPOS.join(res)
```

Notemos que el algoritmo es exponencial ya que para cada elemento realiza k llamadas recursivas y por lo tanto se podría acotar al algoritmo como $O(k^n)$ siendo k el numero de grupos. Para probar el algoritmo resolvimos los casos de la catedra para los cuales el algoritmo dio el resultado correcto aunque para las casos mas grandes el algoritmo no pudo ser probado debido a que su ejecución demoraba demasiado. Además generamos casos aleatorios con la siguiente función para probar el

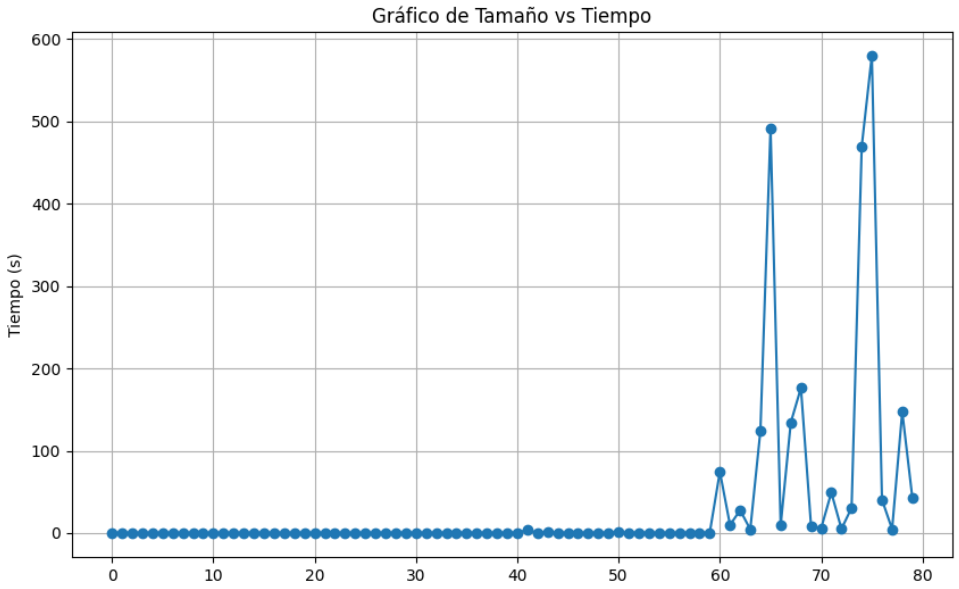
algoritmo y tomar mediciones de tiempo:

```
import random

PRIMER_LINEA = "# #La primera linea indica la cantidad de grupos a formar, las
    siguientes son de la forma 'nombre maestro, habilidad'"
SALTO_DE_LINEA = "\n"


def random_cases_generator(ruta, k):
    indice = 0
    with open(ruta, "w") as archivo:
        archivo.write(PRIMER_LINEA + SALTO_DE_LINEA)
        archivo.write(f"{k}\n")
        suma_grupo = random.randint(600, 3000)
        for i in range(k):
            actual = suma_grupo
            while actual > 0:
                valor = random.randint(int(suma_grupo / 3), suma_grupo)
                if valor > actual:
                    valor = actual
                actual -= valor
                archivo.write(f"{indice}, {valor}" + SALTO_DE_LINEA)
                indice += 1


indice = 0
for i in range(2, 5):
    for n in range(20):
        random_cases_generator(f"casosComparacion/caso{i}_{n}.txt", i)
        indice += 1
```

Los casos estan generados de tal manera que hayan k grupos todos con la misma sumatoria ya que de esta manera sabemos que el coeficiente de la solucion optima es $k * (M/k)^2$ siendo M la sumatoria de todas las habilidades de los maestros. De esta manera podemos comprobar que el algoritmo de backtracking da la solución óptima. Para graficar las mediciones de tiempo usamos el siguiente código:

```
    listaTamanios = []
    listaDuraciones = []
    indice = 0
    for i in range(2, 6):
        for n in range(20):
            caso = f"casosComparacion/caso{i}_{n}.txt"
            k, maestros = leer_archivo(caso)
            inicio = time.time()
            resultado_backtracking = calcular_coeficiente(backtracking(caso))
            fin = time.time()
            assert resultado_backtracking == k * ((sumatoria(maestros) / k) ** 2)
            listaTamanios.append(indice)
            listaDuraciones.append((fin - inicio) * 1000)
            indice += 1
    plt.figure(figsize=(10, 6))
    plt.plot(listaTamanios, listaDuraciones, marker='o', linestyle='-')
    plt.title('Gr fico de Tama o vs Tiempo')
    plt.ylabel('Tiempo (ms)')
    plt.grid(True)
    plt.show()
```

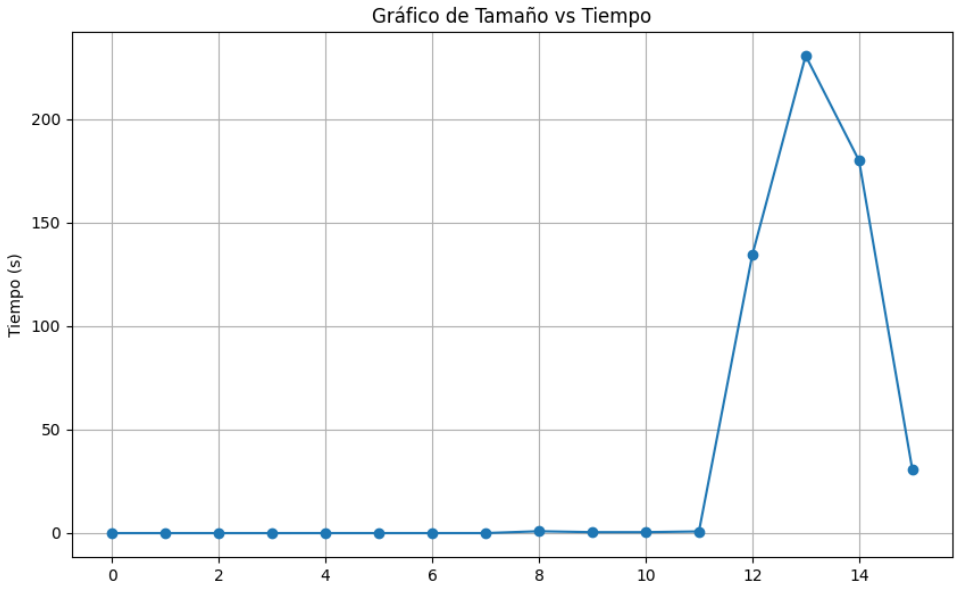
Y los resultados fueron:



Notemos que para los casos de 5 grupos, el algoritmo varió mucho en su tiempo de ejecución esto es debido a las podas aplicadas por el algoritmo que en muchos casos redujo el tiempo de ejecucion del algoritmo. Luego probamos con sets de datos generados de otra manera, la diferencia que para estos sets no se conocía inicialmente la solución óptima y se los usó mas que nada para medir el tiempo que tardaba el algoritmo en resolver un caso totalmente aleatorio. El código para generar estos casos fue:

```
import random

PRIMER_LINEA = "# #La primera linea indica la cantidad de grupos a formar, las
    siguientes son de la forma 'nombre maestro, habilidad'"
SALTO_DE_LINEA = "\n"


def random_cases_generator(ruta, k):
    with open(ruta, "w") as archivo:
        archivo.write(PRIMER_LINEA + SALTO_DE_LINEA)
        archivo.write(f"{k}\n")
        for i in range(k * 3):
            valor = random.randint(50, 400)
            archivo.write(f"{i}, {valor}" + SALTO_DE_LINEA)

for i in range(2, 6):
    for n in range(4):
        random_cases_generator(f"casosComparacion/caso{i}_{n}.txt", i)
```

Finalmente los resultados fueron:

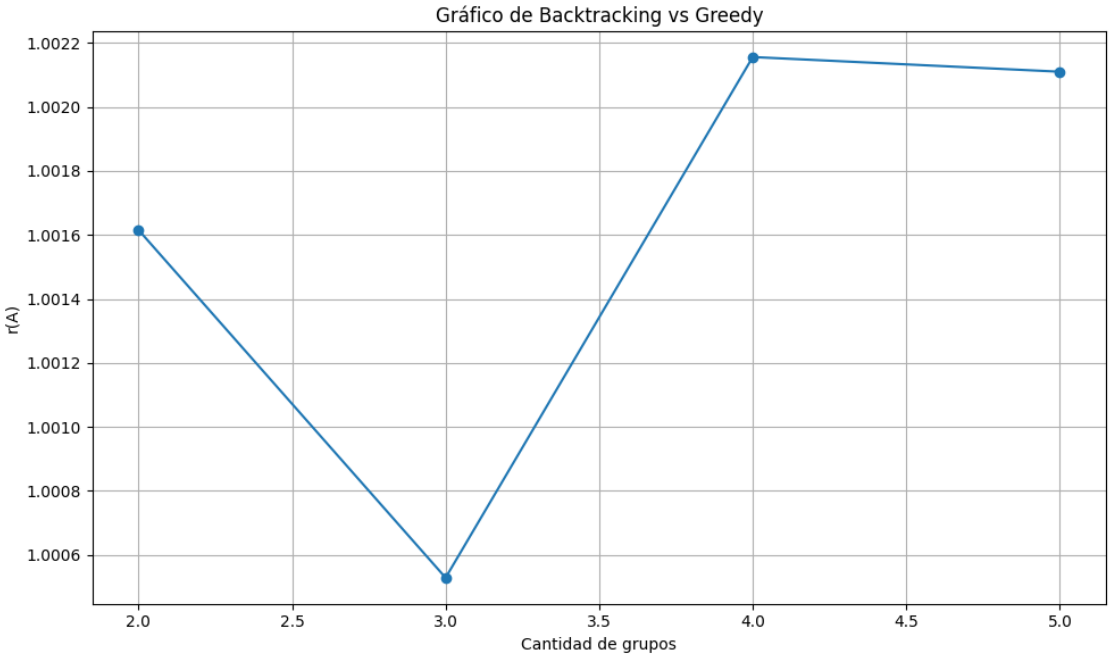


Y notemos que similar al caso anterior, para los sets que se debían separar en 5 grupos, el algoritmo demoró mucho mas en resolverlo y no se aumento k a mas de 5 ya que si no el gráfico no se hacia en un tiempo razonable y podría incluso llegar a demorar varias horas o días en graficar.

Ahora con el algoritmo de backtracking ya presentado, se probó para casos manejables por el algoritmo de backtracking a este último y al algoritmo Greedy presentario anteriormente para ver cuanto variaba la aproximación del algoritmo greedy con la solución óptima. Con casos generados de manera similar a los último se hizo un grafico en el cual r(A) = solución greedy / solución óptima. El código utilizado para realizar el gráfico fue:

```
def graficar_aproximacion_greedy():
    listaTamanios = []
    listaDuraciones = []
    for i in range(2, 6):
        promedio = 0
        for n in range(4):
            caso = f"casosComparacion/caso{i}_{n}.txt"
            k, maestros = leer_archivo(caso)
            resultado_backtracking = calcular_coeficiente(backtracking(caso))
            resultado_greedy = calcular_coeficiente(greedy1(maestros, k))
            promedio += resultado_greedy / resultado_backtracking
        listaTamanios.append(i)
        listaDuraciones.append(promedio / 4)
    plt.figure(figsize=(10, 6))
    plt.plot(listaTamanios, listaDuraciones, marker='o', linestyle='-')
    plt.title('Gr fico de Backtracking vs Greedy')
    plt.xlabel('Cantidad de grupos')
    plt.ylabel('r(A)')
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

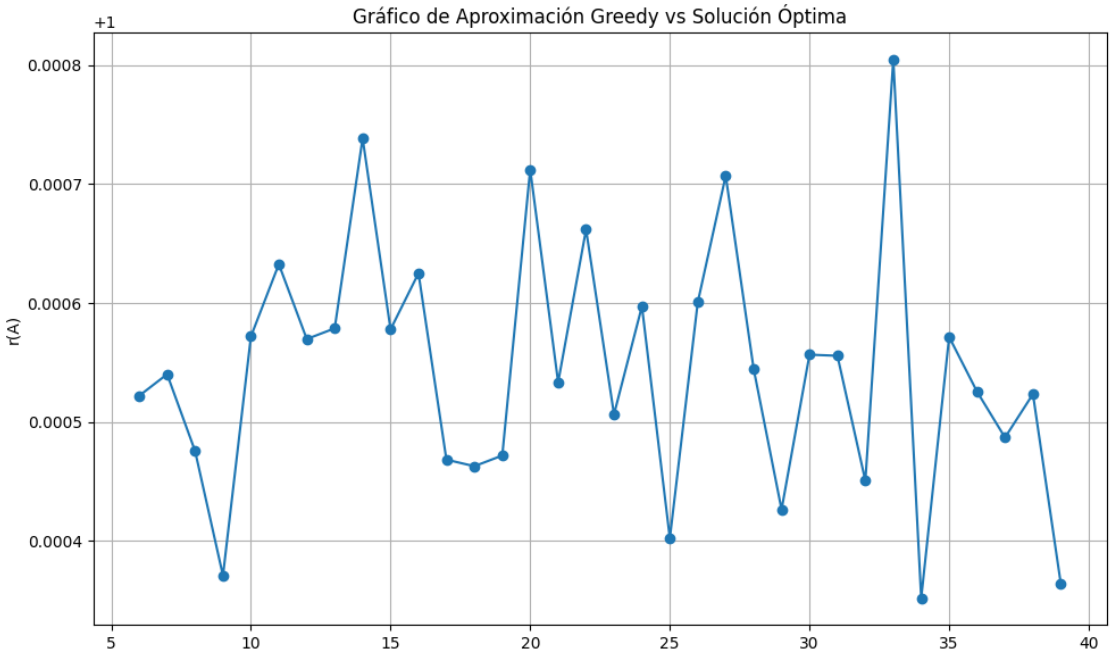
De esta manera para cada k se probaron varios casos y se obtuvo un promedio que sirve para generar una cota más exacta. El gráfico resultante fue:



Notemos que la aproximación se acercó bastante a la solución óptima y se puede acotar a r(A) en 1.0022. Y finalmente se probo el algoritmo greedy para casos ya inmanejables por el algoritmo. Los casos fueron generados de manera que ya se supiese la respuesta óptima de la misma manera que fueron generados los casos para el primer gráfico que se vio. El código para generar el gráfico fue:

```
def graficar_aproximacion_casos_inmanejables_greedy():
    listaTamanios = []
    listaDuraciones = []
    indice = 0
    for i in range(6, 40):
        promedio = 0
        for n in range(20):
            caso = f"casosInmanejables/caso{indice}.txt"
            k, maestros = leer_archivo(caso)
            resultado = k * ((sumatoria(set(maestros)) / k) ** 2)
            resultado_greedy = calcular_coeficiente(greedy1(maestros, k))
            promedio += resultado_greedy / resultado
            indice += 1
        listaDuraciones.append(promedio / 20)
        listaTamanios.append(i)
    plt.figure(figsize=(10, 6))
    plt.plot(listaTamanios, listaDuraciones, marker='o', linestyle='-')
    plt.title('Gr fico de Aproximaci n Greedy vs Soluci n  ptima ')
    plt.ylabel('r(A)')
    plt.grid(True)
    plt.tight_layout()
    plt.show(=
```

El gráfico generado fue:



Notemos que de manera similar al gráfico anterior, r(A) es menor a 1.0022 para todos los casos.

5. Aproximación por Programación Lineal

Para resolver el problema de distribución de maestros de agua en k grupos de manera equitativa, utilizamos un modelo de programación lineal. Nuestro objetivo es minimizar la diferencia entre las sumas de habilidades del grupo con mayor suma y el grupo con menor suma. Dado que la función objetivo original (minimizar la suma de los cuadrados de las sumas de las fuerzas de los grupos) es cuadrática, planteamos alternativas que permiten obtener una solución aproximada.

Formulación del Problema

Variables

- $y_{ij} \in \{0, 1\}$: Variable binaria que indica si el maestro i está en el grupo j.
- S_j: Fuerza total del grupo j.
- S_{max}: Fuerza total del grupo con la mayor suma de habilidades.
- S_{min}: Fuerza total del grupo con la menor suma de habilidades.

Función Objetivo

Minimizar la diferencia entre la suma de habilidades del grupo con mayor suma y el grupo con menor suma:

$$\text{Minimizar} \quad S_{max} - S_{min}$$

Restricciones

Cada maestro debe estar en exactamente un grupo:

$$\sum_{j=1}^{k} y_{ij} = 1 \quad \forall i \in \{1, \ldots, n\}$$

La fuerza total de cada grupo j se define como la suma de las habilidades de los maestros en ese grupo:

$$S_j = \sum_{i=1}^{n} x_i \cdot y_{ij} \quad \forall j \in \{1, \ldots, k\}$$

Definición de S_{max} y S_{min}:

$$S_{max} \geq S_j \quad \forall j \in \{1, \ldots, k\}$$
$$S_{min} \leq S_j \quad \forall j \in \{1, \ldots, k\}$$

```python
def pl(maestros, k):

    modelo = pulp.LpProblem("DistribucionMaestrosAgua", pulp.LpMinimize)

    y = pulp.LpVariable.dicts("y", ((i, j) for i in range(len(maestros)) for j in range(k)), cat=pulp.LpBinary)
    S = pulp.LpVariable.dicts("S", range(k), lowBound=0, cat=pulp.LpContinuous)
    S_max = pulp.LpVariable("S_max", lowBound=0, cat=pulp.LpContinuous)
    S_min = pulp.LpVariable("S_min", lowBound=0, cat=pulp.LpContinuous)

    modelo += S_max - S_min

    for i in range(len(maestros)):
        modelo += pulp.lpSum(y[i, j] for j in range(k)) == 1

    for j in range(k):
        modelo += S[j] == pulp.lpSum(maestros[i] * y[i, j] for i in range(n))

    for j in range(k):
        modelo += S_max >= S[j]
        modelo += S_min <= S[j]

    modelo.solve()

    if modelo.status == pulp.LpStatusOptimal:
        grupos = []
        for j in range(k):
            grupo = [i for i in range(n) if pulp.value(y[i, j]) > 0.5]
            grupos.append(grupo)
        diferencia_minima = pulp.value(S_max) - pulp.value(S_min)
        return grupos, diferencia_minima
    else:
        return None, None
```

El objetivo de este modelo es dividir un grupo de maestros agua en grupos distintos de modo que la suma de las habilidades de los grupos con la suma más alta se mantenga constante mientras que la suma de los grupos con la suma más baja se mantenga constante.

El modelo emplea variables de decisión para determinar si cada maestro está asignado a un grupo en particular. La asignación de cada maestro a un grupo está indicada por estas variables binarias.

Las restricciones del modelo garantizan que cada maestro sea asignado a un grupo específico y que la variable suma en el modelo sea igual a la variable suma en la variable suma. Además, se establecen limitaciones para garantizar que las variables de suma máxima y mínima sean iguales y superen la suma de habilidades de cada grupo, respectivamente.

Una vez que se satisfacen todas las variables y restricciones, el modelo se resuelve para determinar la forma más eficiente de asignar maestros a los grupos manteniendo al mínimo la diferencia entre las sumas de habilidades.

Diferencias de tiempo entre el algoritmo de Backtracking y Programación Lineal

Ejemplo de la Catedra "$11 - 5.txt$"

Tiempo de minimizar la diferencia del grupo de mayor suma con el de menor suma (Programación lineal):

Total time (CPU seconds): 0.97

Diferencia mínima: 218.0

Tiempo empleado en minimizar la adición de los cuadrados de las sumas de las fuerzas de los grupos (Backtracking) :

Backtracking: 0.24 segundos

Ejemplo de la Catedra "$14 - 3.txt$"

Total time (CPU seconds): 3.07

Diferencia mínima: 1.0

Backtracking: 2.20 segundos

Ejemplo de la Catedra "$15 - 4.txt$"

Programación Lineal: Total time (CPU seconds): 12.19

Diferencia mínima: 7.0

Backtracking: Execution time: 6.2371485233306885 seconds

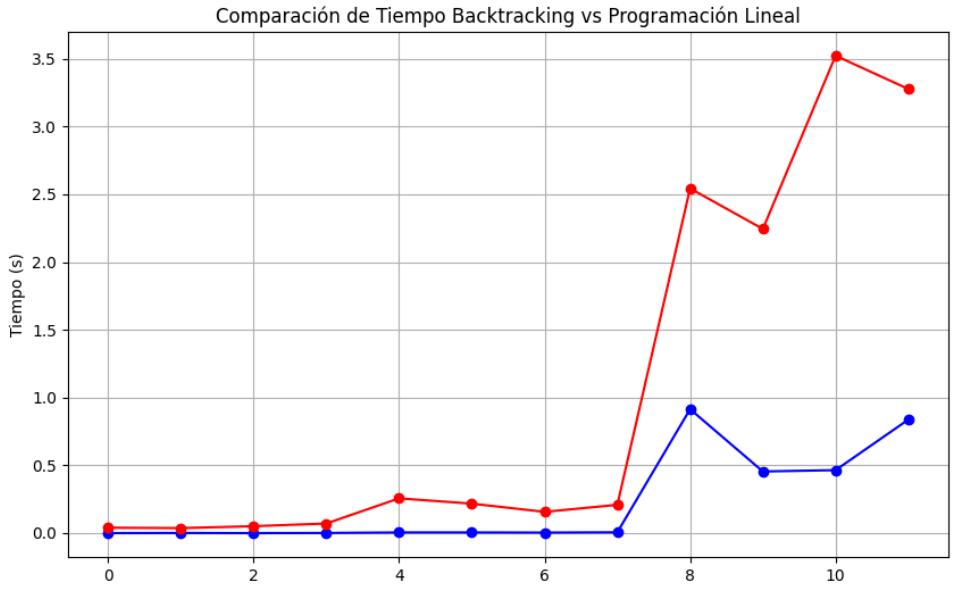
5.1. Ejemplos Aleatorios

También se probó a ambos algoritmos con los casos generados anteriormente para realizar mediciones de tiempo para el algoritmo de backtracking, esta vez se tomaron mediciones de tiempo para ambos algoritmos y también se calculó r(A) de la aproximación por programación lineal. Para tomar mediciones de tiempo se utilizo el siguiente código:

```
def graficar_tiempo_aproximacion_pl():
    listaTamanios_bt = []
    listaDuraciones_bt = []
    listaTamanios_pl = []
    listaDuraciones_pl = []
    indice = 0
    for i in range(2, 5):
        for n in range(4):
            caso = f"casosComparacion/caso{i}_{n}.txt"
            k, maestros = leer_archivo(caso)
            inicio = time.time()
            calcular_coeficiente(backtracking(caso))
            fin = time.time()
            listaDuraciones_bt.append(fin - inicio)
            listaTamanios_bt.append(indice)
            inicio = time.time()
            calcular_coeficiente(pl(maestros, k))
            fin = time.time()
            listaDuraciones_pl.append(fin - inicio)
```

```
            listaTamanios_pl.append(indice)
            indice += 1
    plt.figure(figsize=(10, 6))
    plt.plot(listaTamanios_bt, listaDuraciones_bt, marker='o', linestyle='-', color
    ="blue")
    plt.plot(listaTamanios_pl, listaDuraciones_pl, marker='o', linestyle='-', color
    ="red")
    plt.title('Comparaci n de Tiempo Backtracking vs Programaci n Lineal')
    plt.ylabel('Tiempo (s)')
    plt.grid(True)
    plt.show()
```

El gráfico generado fue el siguiente:



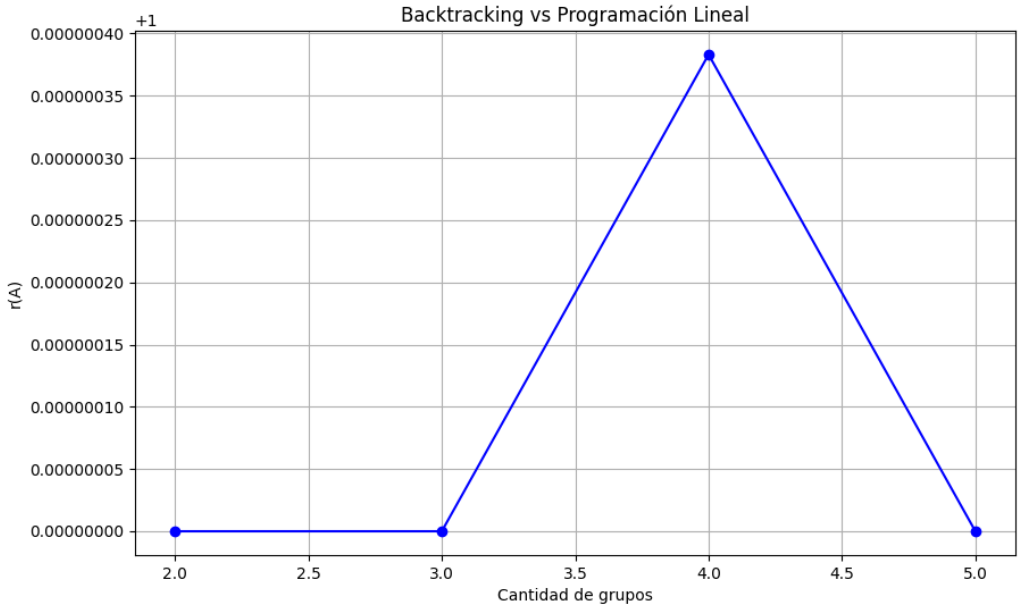
La función roja corresponde a la aproximación y la azul al algoritmo de backtracking y se puede ver a simple vista que para casos pequeños ambos algoritmos demoran similar pero para casos mas grandes el algoritmo de programación lineal comienza a demorar significativamente más que el de backtracking, esto se puede deber a que el algoritmo de backtracking tiene varias podas que lo vuelven más veloz.

Y finalmente para gráficar r(A) se utilizo el siguiente código que utiliza los mismos casos que se utilizaron para las mediciones de tiempo:

```
def graficar_aproximacion_pl():
    listaTamanios = []
    listaDuraciones = []
    indice = 0
    for i in range(2, 6):
        promedio = 0
        for n in range(4):
            caso = f"casosComparacion/caso{i}_{n}.txt"
            k, maestros = leer_archivo(caso)
            sol_optima = calcular_coeficiente(backtracking(caso))
            sol_pl = calcular_coeficiente(pl(maestros, k))
            indice += 1
            promedio += sol_pl / sol_optima
        listaTamanios.append(i)
        listaDuraciones.append(promedio / 4)
    plt.figure(figsize=(10, 6))
    plt.plot(listaTamanios, listaDuraciones, marker='o', linestyle='-', color="blue
    ")
    plt.title('Backtracking vs Programaci n Lineal')
    plt.xlabel('Cantidad de grupos')
```

```
    plt.ylabel('r(A)')
    plt.grid(True)
    plt.gcf().axes[0].yaxis.get_major_formatter().set_scientific(False)
    plt.show()
```

El gráfico resultante fue:



Notemos que generalmente la aproximación fue igual a la solución óptima y r(a) puede acotarse por 1.00000040.

6. Algoritmo Extra

Como última aproximación se nos ocurrió un algoritmo utilizando programación lineal entera:

6.1. Formulación del Problema

Lo primero que hicimos fue plantear las variables usadas por el algoritmo:

Variables

- $y_{ij} \in \{0, 1\}$: Variable binaria que indica si el maestro i está en el grupo j.
- S_j: Fuerza total del grupo j.
- S_{max}: Fuerza total del grupo con la mayor suma de habilidades.

Función Objetivo

Minimizar la suma de habilidades del grupo con mayor suma:

$$\text{Minimizar} \quad S_{max}$$

Restricciones

Cada maestro debe estar en exactamente un grupo:

$$\sum_{j=1}^{k} y_{ij} = 1 \quad \forall i \in \{1, \ldots, n\}$$

La fuerza total de cada grupo j se define como la suma de las habilidades de los maestros en ese grupo:

$$S_j = \sum_{i=1}^{n} x_i \cdot y_{ij} \quad \forall j \in \{1, \ldots, k\}$$

Definición de S_{max}:

$$S_{max} \geq S_j \quad \forall j \in \{1, \ldots, k\}$$

El código del algoritmo en cuestión:

```python
import pulp


def aproximacion_pl(maestros, k):
    problema = pulp.LpProblem("problema", pulp.LpMinimize)
    grupos = range(k)
    res = []
    for g in grupos:
        res.append(set())
    y = pulp.LpVariable.dict("maestros", (grupos, maestros), cat=pulp.LpBinary)
    for maestro in maestros:
        ecuacion = 0
        for g in grupos:
            ecuacion += y[(g,maestro)]
        problema += ecuacion == 1
    sumatoria = []
    for g in grupos:
        sumatoria.append(pulp.lpSum([y[(g, maestro)] * maestro[1] for maestro in
    maestros]))
    maximo = pulp.LpVariable("GRUPO_MAX", cat=pulp.LpInteger)
    for suma in sumatoria:
        problema += maximo >= suma
    problema += maximo
    problema.solve()
    for maestro in maestros:
        for g in grupos:
            if pulp.value(y[(g, maestro)]) > 0:
                res[g].add(maestro)
    return res
```

Este algoritmo es similar al anterior solo que en este caso la función objetivo en lugar de buscar minimizar la diferencia entre la suma del grupo con mayor suma de habilidades y el de menor suma, únicamente busca minimizar la suma del grupo con mayor suma de habilidades. Y como para minimizar al grupo de suma máxima el algoritmo debe distribuir lo mejor posible los grupos de manera que el grupo de mayor suma tenga la menor suma posible y tomamos a esta distribución como una aproximación a la solución óptima. Como el algoritmo utiliza programación lineal entera, su complejidad es exponencial. Como el algoritmo funciona con programación lineal entera, lo comparamos con el algoritmo presentado anteriomente y también con el algoritmo Greedy. Para ambos casos usaremos los mismos casos que utilizamos para el algoritmo anterior.

6.2. Comparación con el primer algoritmo de programación lineal

Primero se compararon las aproximaciones dadas por ambos algoritmos, el código utilizado para generar el gráfico fue el siguiente:

```python
def graficar_aproximaciones_pl():
    listaTamanios_pl1 = []
    listaDuraciones_pl1 = []
    listaTamanios_pl2 = []
    listaDuraciones_pl2 = []
    indice = 0
    for i in range(2, 5):
        for n in range(4):
            caso = f"casosComparacion/caso{i}_{n}.txt"
            k, maestros = leer_archivo(caso)
            habilidades = [habilidad for nombre, habilidad in maestros]

            grupos, diferencia_minima = pl(habilidades, k)
            diccionario = crear_diccionario_maestros(maestros)
            valores = []
            for g in grupos:
                nuevo = set()
                for elem in g:
                    nuevo.add((elem, diccionario[elem]))
                valores.append(nuevo)
            sol_pl1 = calcular_coeficiente(valores)
            sol_pl2 = calcular_coeficiente(aproximacion.aproximacion_pl(maestros, k
    ))
            indice += 1
            listaTamanios_pl1.append(indice)
            listaTamanios_pl2.append(indice)
            listaDuraciones_pl1.append(sol_pl1)
            listaDuraciones_pl2.append(sol_pl2)
    plt.figure(figsize=(10, 6))
    plt.plot(listaTamanios_pl1, listaDuraciones_pl1, marker='o', linestyle='-',
    color="blue")
    plt.plot(listaTamanios_pl2, listaDuraciones_pl2, marker='o', linestyle='-',
    color="red")
    plt.title('Comparaci n resultados algoritmos por programaci n lineal')
    plt.xlabel('Cantidad de grupos')
    plt.ylabel('Coeficiente')
    plt.grid(True)
    plt.show()
```

Y el gráfico obtenido fue:

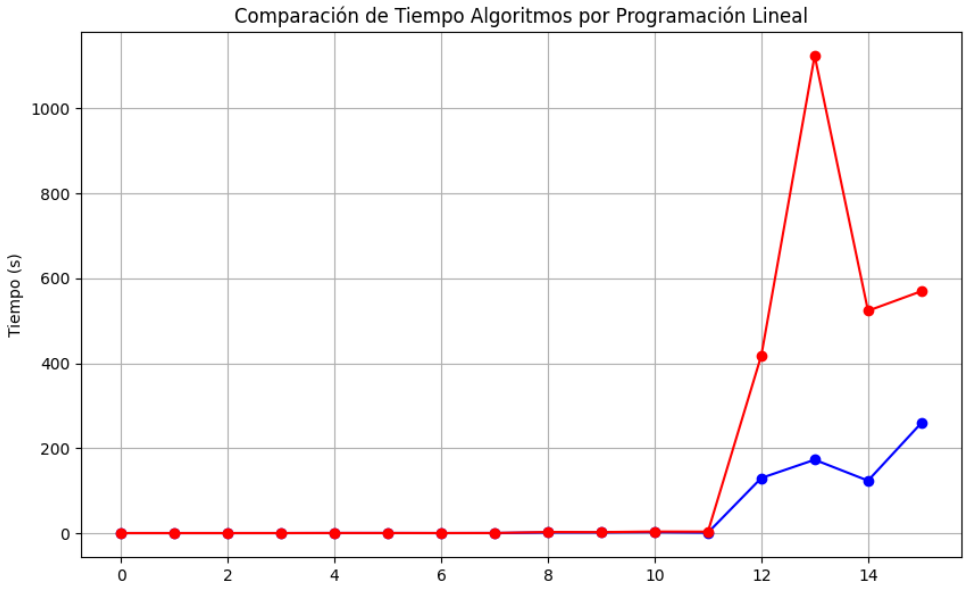
Notemos que ambos algoritmos dieron la misma aproximación para todos los casos.

Luego se realizaron mediciones de tiempo para ambos algoritmos y el código para realizar el

grafico fue:

```
def graficar_tiempo_aproximaciones_pl():
    listaTamanios_pl2 = []
    listaDuraciones_pl2 = []
    listaTamanios_pl1 = []
    listaDuraciones_pl1 = []
    indice = 0
    for i in range(2, 6):
        for n in range(4):
            caso = f"casosComparacion/caso{i}_{n}.txt"
            k, maestros = leer_archivo(caso)
            inicio = time.time()
            aproximacion.aproximacion_pl(maestros, k)
            fin = time.time()
            listaDuraciones_pl2.append(fin - inicio)
            listaTamanios_pl2.append(indice)
            habilidades = [habilidad for nombre, habilidad in maestros]
            inicio = time.time()
            pl(habilidades, k)
            fin = time.time()
            listaDuraciones_pl1.append(fin - inicio)
            listaTamanios_pl1.append(indice)
            indice += 1
    plt.figure(figsize=(10, 6))
    plt.plot(listaTamanios_pl2, listaDuraciones_pl2, marker='o', linestyle='-',
    color="blue")
    plt.plot(listaTamanios_pl1, listaDuraciones_pl1, marker='o', linestyle='-',
    color="red")
    plt.title('Comparaci n de Tiempo Algoritmos por Programaci n Lineal')
    plt.ylabel('Tiempo (s)')
    plt.grid(True)
    plt.show()
```

Y los resultados fueron:



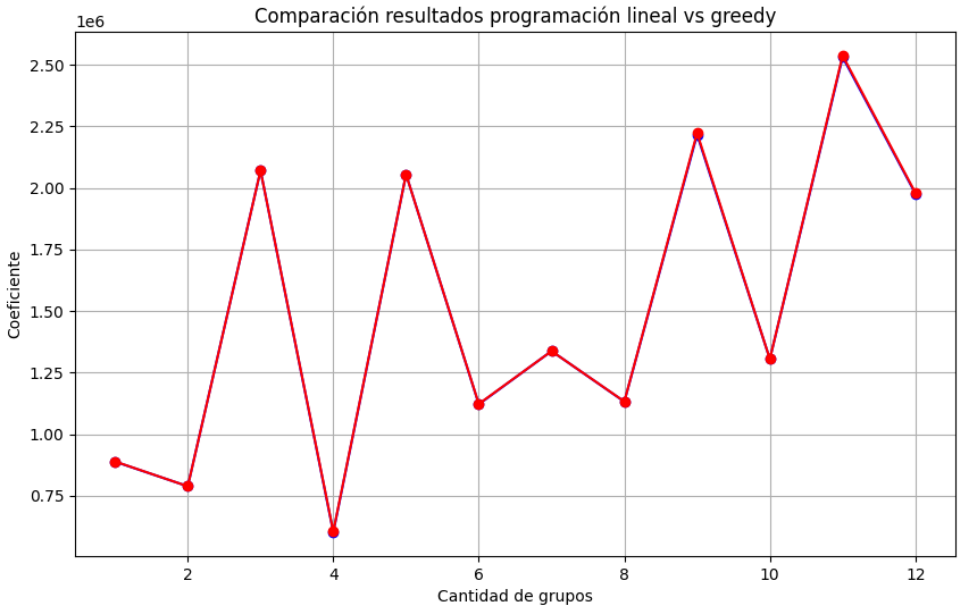
Notemos que para los casos mas pequeños ambos algoritmos toman un tiempo similar pero para casos más grandes el algoritmo anterior (función roja) demora significativamente más que el algoritmo actual por programación lineal.

6.3. Comparación con el primer algoritmo greedy

Se compararon las aproximaciones dadas por ambos algoritmos y se realizó un gráfico con los resultados de ambos algoritmos, el código utilizado fue:

```
def graficar_aproximaciones_vs_greedy():
    listaTamanios_pl = []
    listaDuraciones_pl = []
    listaTamanios_greedy = []
    listaDuraciones_greedy = []
    indice = 0
    for i in range(2, 5):
        for n in range(4):
            caso = f"casosComparacion/caso{i}_{n}.txt"
            k, maestros = leer_archivo(caso)
            sol_pl = calcular_coeficiente(aproximacion.aproximacion_pl(maestros, k)
    )
            sol_greedy = calcular_coeficiente(greedy1(maestros, k))
            indice += 1
            listaTamanios_pl.append(indice)
            listaTamanios_greedy.append(indice)
            listaDuraciones_pl.append(sol_pl)
            listaDuraciones_greedy.append(sol_greedy)
    plt.figure(figsize=(10, 6))
    plt.plot(listaTamanios_pl, listaDuraciones_pl, marker='o', linestyle='-', color
    ="blue")
    plt.plot(listaTamanios_greedy, listaDuraciones_greedy, marker='o', linestyle='-
    ', color="red")
    plt.title('Comparaci n resultados algoritmos por programaci n lineal')
    plt.xlabel('Cantidad de grupos')
    plt.ylabel('Coeficiente')
    plt.grid(True)
    plt.show()
```

Los resultados fueron:

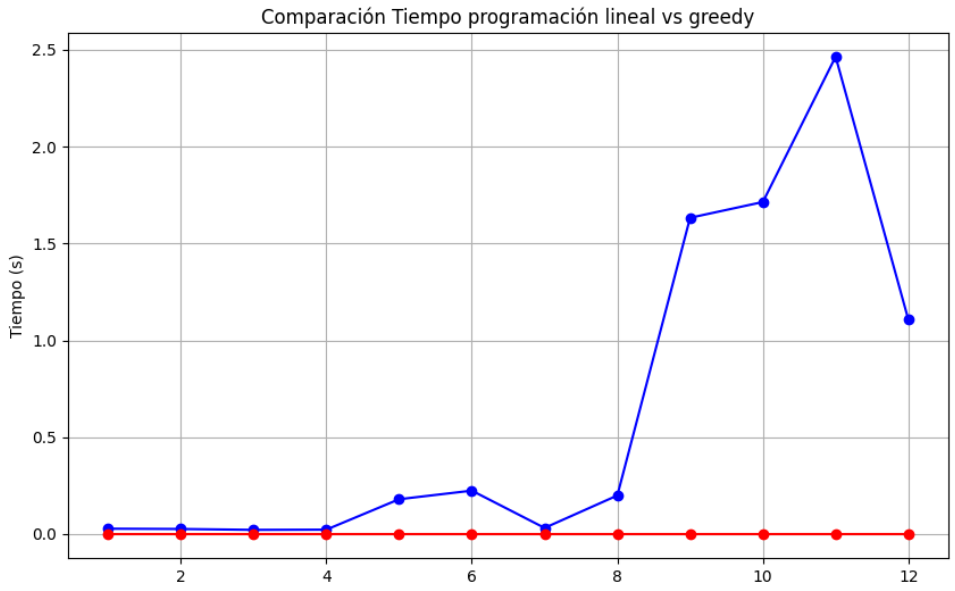


Notemos que ambos algoritmos dieron la misma aproximación, pero el algoritmo greedy lo hizo demorando mucho menos tiempo como se verá en el gráfico a continuación en el que se midió el tiempo que tardó cada uno. El código utilizado para graficar los resultados fue:

```
def graficar_tiempo_aproximaciones_vs_greedy():
    listaTamanios_pl = []
    listaDuraciones_pl = []
    listaTamanios_greedy = []
```

```
    listaDuraciones_greedy = []
    indice = 0
    for i in range(2, 5):
        for n in range(4):
            caso = f"casosComparacion/caso{i}_{n}.txt"
            k, maestros = leer_archivo(caso)
            ini = time.time()
            aproximacion.aproximacion_pl(maestros, k)
            fin = time.time()
            listaDuraciones_pl.append(fin - ini)
            ini = time.time()
            greedy1(maestros, k)
            fin = time.time()
            listaDuraciones_greedy.append(fin - ini)
            indice += 1
            listaTamanios_pl.append(indice)
            listaTamanios_greedy.append(indice)
    plt.figure(figsize=(10, 6))
    plt.plot(listaTamanios_pl, listaDuraciones_pl, marker='o', linestyle='-', color
    ="blue")
    plt.plot(listaTamanios_greedy, listaDuraciones_greedy, marker='o', linestyle='-
    ', color="red")
    plt.title('Comparaci n Tiempo programaci n lineal vs greedy')
    plt.ylabel('Tiempo (s)')
    plt.grid(True)
    plt.show()
```

El resultado fue:



Se puede observar que el algoritmo greedy (función roja) concluyó mucho mas rapido que el algoritmo por programación lineal y dió la misma aproximación.

7. Conclusiones

En conclusión, después de analizar el problema, tanto el algoritmo de backtracking como el de programación lineal, concluimos a que se ha encontrado una solución óptima para hallar la solución antes mencionada. En cuanto a los tiempos de ejecución, en Backtracking se puede mencionar que, gracias a las podas de casos en los que ya no se encontraría la solución, se logró llegar a un algoritmo que ejecuta con una rapidez considerable los casos propuestos por la cátedra. Se logró demostrar que el problema planteado corresponde a un problema NP-Completo.

A través de la implementación de un modelo de programación lineal para resolver el problema de distribución equitativa de maestros de agua en k grupos, hemos logrado minimizar la diferencia entre las sumas de habilidades del grupo con mayor suma y el grupo con menor suma. Los resultados obtenidos de la comparación entre el algoritmo de backtracking y la programación lineal muestran diferencias significativas en términos de tiempo de ejecución y precisión de las soluciones.

Por último, hemos demostrado y verificado dos métodos distintos para resolver el problema de la distribución maestra del agua. Ambos métodos han sido probados y utilizados con éxito en diversos casos, ofreciendo soluciones que cumplen con los criterios establecidos. El tamaño, la complejidad y los recursos disponibles del problema determinarán qué método es mejor para el problema. Estamos satisfechos con los resultados,