Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

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

06-05-2024

Álvarez Julián 110825

Ascencio Felipe Santino 110675

Olivera Nicolás 110797

Introducción

1. Análisis del enunciado

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 (S1, S2, ..., Sk).

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 "xi", y tenemos todos los valores "x1, x2, ..., xn"(todos son 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.

2. Análisis problema

El problema presenta un escenario estratégico donde se deben optimizar las decisiones de las distribuciones del ataque para minimizar la ecuación planteada.

Es decir, el problema se basa en encontrar la mejor distribución de maestros agua, para cumplir con el enunciado de forma óptima.

En este caso nos piden utilizar 3 algoritmos:

El primero de Backtracking, el cual nos va a dar una solución óptima al problema.

El segundo es un Modelo de programación lineal.

El tercero es un Algoritmo de aproximación, el cual nos dará, como bien dice su nombre, un valor aproximado del cual obtendremos su cota empírica de aproximación.

En este informe se pasa a detallar el paso a paso seguido para encontrar los algoritmos, y la explicación del funcionamiento de cada uno, que resuelva el enunciado de forma óptima o aproximada según corresponda.

3. Pequeña consideración

La explicación de la distribución de los archivos de "GITHUB", los resultados de las pruebas propias y aproximaciones creadas, y la explicación de ejecución de los programas se encuentran en el "README" del repositorio.

Desarrollo

4. Demostración de que el problema es NP-Completo

Antes de plantear las resoluciones óptimas al enunciado, primero vamos a pasar a analizar y demostrar que el problema es "NP-Completo".

Para demostrar que el Problema de la Tribu del Agua es NP-completo, seguimos estos pasos:

Mostraremos que el problema está en NP: Esto significa que dada una partición de los maestros agua en k subgrupos y un límite B, se puede verificar en tiempo polinómico si la partición cumple con las condiciones del problema.

Seleccionaremos un problema conocido NP-completo: Encontramos un problema que ya se haya demostrado que está en NP y que sea NP-completo.

Realizamos una reducción polinómica: Debemos demostrar que se puede convertir una instancia del Problema NP completo elegido en una instancia equivalente del Problema de la Tribu del Agua en tiempo polinómico.

5. Demostración de que el Problema de la Tribu del Agua está en NP

Para demostrar que el Problema de la Tribu del Agua está en NP, debemos verificar que dado un certificador (una partición en k subgrupos), podemos verificar en tiempo polinómico si este certificador es válido. Dado un validador que consiste en una partición en k subgrupos $S1, S2, ..., Sk$ verificamos si:

1 : Cada "xi" está asignado a un grupo y solo a un grupo.

2 : La suma de los cuadrados de las sumas de las fuerzas de cada subgrupo es menor o igual a B.

Ambas verificaciones pueden realizarse en tiempo polinómico, por lo tanto, el Problema de la Tribu del Agua está en NP.

```
1
2  def verificar_resultado(particion, k, B, maestros):
3
4
5      xi_en_grupo = set()
6      sum_total = 0
7      # Verificar si cada xi est  asignado solo a un grupo
8      for grupo in particion:
9          suma_fuerzas_subgrupo = 0
10         for xi in grupo:
11             if xi in xi_en_grupo:
12                 return False
13             xi_en_grupo.add(xi)
14             suma_fuerzas_subgrupo += xi
15
16         sum_total += suma_fuerzas_subgrupo ** 2
17
18     if xi_en_grupo!= maestros:
19         return False
20
21
22     if sum_total > B:
23         return False
24
25     return True
```

En la función verificar-resultado, se recorren todos los elementos (xi) en todos los grupos de la partición utilizando un único bucle for. Dado que hay n elementos en total (donde n es la suma

de los elementos en todos los grupos), la complejidad de recorrer todos estos elementos es O(n). Y teniendo en cuenta que la complejidad de agregar un elemento a un conjunto (set) en Python es O(1) en promedio, entonces la complejidad computacional total de la función verificar-certificado es O(n)). Maestros es un set que contiene todo el conjunto de maestros a colocar en grupos, si alguno no esta en xi-en-grupo entonces la solucion particular es invalida.

6. Demostración de que el Problema de la Tribu del Agua es NP-Completo

Para demostrar que el Problema de la Tribu del Agua es NP-completo, haremos una reducción polinomial desde un problema conocido que es NP-completo. Utilizaremos el Problema de Partición, que es un problema NP-completo clásico.

Para recordar, el Problema de la Tribu del Agua se define como:

Entrada: Un conjunto de números x1,x2,...,xn (representando las fuerzas de los maestros), un número k de grupos y un valor B que representa la restricción del problema.

Pregunta: ¿Podemos asignar estos números a k grupos de manera que la suma de los cuadrados de las sumas de las fuerzas de los grupos sea menor o igual a B?

Reducción desde el Problema de Partición:

1: Entrada del Problema de Partición: Un conjunto de números $A = a1, a2, \ldots, an$ y un valor V que representa una suma de un subconjunto de A.

2: Transformación a la entrada del Problema de la Tribu del Agua:

- Definimos $k = 2$ (queremos dividir el conjunto en 2 grupos).
- Asignamos las fuerzas de los maestros como los elementos del conjunto A, es decir, "$xi = ai$".
- Definimos B como la sumatoria de todas las fuerzas de los maestros dividido dos y elevado al cuadrado. ($2V^2$) (ver más abajo la explicacion del porque). siendo V la suma de cada uno de los dos subconjuntos.

3: Solución del Problema de Partición: El problema pregunta si se puede dividir "A" en dos subconjuntos con la misma suma "V" cada uno.

Caja negra del Problema de la Tribu del Agua

Esta caja negra toma el conjunto de fuerzas "x1,x2,...,xn", $k = 2$ y $B = 2V^2$, y devuelve si existe una solución que sea menor o igual a B. Es decir, cuando podemos dividir el conjunto en 2 grupos de manera que la suma de cuadrados de las sumas de las fuerzas de cada grupo sea inferior o igual a un valor B: $V^2 + V^2 <= B$. Y devuelve False cuando no existe tal solución.

Queremos saber si podemos dividir A en dos subconjuntos con la misma suma V. Esto es equivalente a preguntarle a la caja negra del Problema de la Tribu del Agua si podemos asignar los elementos de A a 2 grupos de manera que se cumpla la restriccion brindada por ($B = 2V^2$).

Para formalizar esto, consideremos que tenemos una función tribu-del-agua(A,k,B) que toma un conjunto A, un número k y un valor B, y devuelve True o false unicamente.

Para demostrar que esta reducción es válida, mostramos que si tribu-del-agua(A,2,$2V^2$)devuelve True, entonces A puede ser dividido en dos subconjuntos de sumas iguales a V.

¿Por que elegimos ese valor de B?

Consideremos dos grupos "S1τ "S2çon las sumas de fuerzas "S1τ "S2respectivamente. Queremos minimizar:

$$\left(\sum_{x_j \in S_1} x_j\right)^2 + \left(\sum_{x_j \in S_2} x_j\right)^2$$

Supongamos que la suma total de las fuerzas es T. Entonces tenemos:

$\left(\sum_{x_j \in S_1} x_j\right)^2 = \text{S1}$

$\left(\sum_{x_j \in S_2} x_j\right)^2 = \text{S2}$

Con la condición de que: S1 + S2 = T

Queremos minimizar $S_1^2 + S_2^2$. Usando la identidad algebraica:

$S_1^2 + S_2^2 = \left(\frac{S_1+S_2}{\sqrt{2}}\right)^2 + \left(\frac{S_1-S_2}{\sqrt{2}}\right)^2$

Dado que S1+S2=T, esto se convierte en:

$S_1^2 + S_2^2 = \left(\frac{T}{\sqrt{2}}\right)^2 + \left(\frac{S_1-S_2}{\sqrt{2}}\right)^2$

El término $\left(\frac{T}{\sqrt{2}}\right)^2$ es constante para un T dado, mientras que el término $\left(\frac{S_1-S_2}{\sqrt{2}}\right)^2$ es cero si y sólo si $S1 = S2 = V$.

En este caso, como se trata de resolver el problema de partition, la caja negra del problema de la tribu del agua tratara de armar dos grupos de sumas iguales y preguntará si es menor a $\left(\frac{T}{\sqrt{2}}\right)^2 = \left(\frac{2V}{\sqrt{2}}\right)^2 = 2V^2$. Devolviendo true o false según corresponda.

Por lo tanto, para minimizar la función objetivo $S_1^2 + S_2^2$, S1 y S2 deben ser lo más iguales posible, es decir, "S1 = S2".

Es por ese motivo que pudimos relacionarlo con el problema de partition y realizar la reduccion polinomial.

7. Reducción inversa

Transformación de Entrada:

1: Tomamos una instancia del Problema de la Tribu del Agua con un conjunto x1,x2,. . . ,xn , k = 2 y con cierto valor de B.

2: Creamos un conjunto A = x1,x2,. . . ,xn para usar como entrada en el Problema de Partición, el valor V saldra del despeje de la igualdad planteada anteriormente: B = $2V^2$ $x \to y$ V = $\sqrt{\frac{B}{2}}$

Uso de la Caja Negra de Partición:

1: Llamamos a la caja negra del Problema de Partición con la entrada A.

2: La caja negra devuelve True si es posible dividir A en dos subconjuntos con la misma suma $\sqrt{\frac{B}{2}}$, y False si no es posible.

Interpretación de Resultados

1: Si la caja negra devuelve True:

- Esto significa que podemos dividir A en dos subconjuntos con la misma suma V.
- Por lo tanto, hemos encontrado una asignación que cumple con la restriccion para la funcion del Problema de la Tribu del Agua para k = 2.

2: Si la caja devuelve False:

- Esto significa que no podemos dividir A en dos subconjuntos con la misma suma.
- En este caso, no podemos garantizar que la función objetivo esté minimizada de manera óptima.

8. Extensión a $k > 2$

Para $k > 2$, la situación se vuelve más complicada. Vamos a analizar por qué la misma lógica no se aplica directamente:

Problema de la Tribu del Agua para $k > 2$:

Ahora tenemos que asignar los números a k grupos de manera que podamos cumplir con la restriccion dada por B.

Diferencias con k = 2k:

- **Equilibrio en más de dos grupos:** Minimizar la suma de los cuadrados de las sumas de más de dos grupos no necesariamente implica que las sumas deban ser iguales. Podrían existir múltiples distribuciones óptimas que no se traducen directamente a particiones con sumas iguales.
- La relación entre las sumas de los grupos es más compleja y no se puede expresar tan directamente.

Complejidad adicional:

Para k¿2, la solución óptima podría tener algunos grupos con sumas significativamente diferentes, lo que no ayuda a resolver el Problema de Partición de manera directa.

9. Explicación algoritmo Backtracking

A continuación se pasa a detallar el paso a paso del funcionamiento del algoritmo.

9.1. Paso 1: Carga de Datos

El programa comienza solicitando al usuario el nombre del archivo que contiene los datos de entrada.

Luego, se lee el archivo y se cargan los datos en la función carga.

Se asume que los datos están en un archivo de texto, donde la primera línea indica el número de grupos a formar y las líneas siguientes contienen los nombres de los maestros agua y sus respectivos poderes.

9.2. Paso 2: Solución Greedy

Antes de realizar el backtracking, se aplica un enfoque greedy (voraz) para obtener una solución inicial. En la función "sol-Greedy", los maestros agua se ordenan por su poder de manera descendente y se asignan a los grupos de manera que el grupo con menor suma de poderes tenga el maestro agua más poderoso disponible.

Esta solución greedy se utiliza como punto de partida para el backtracking.

9.3. Paso 3: Backtracking

El backtracking se realiza en la función "Backtracking".

Esta función utiliza una estrategia de búsqueda exhaustiva para encontrar la combinación óptima de grupos que minimice la suma de los cuadrados de los poderes.

Se exploran todas las posibles combinaciones de asignación de maestros agua a los grupos mediante un proceso recursivo.

Criterio de Parada: Si la suma actual de los cuadrados de los poderes es mayor o igual a la mejor suma obtenida hasta el momento, o si no hay suficientes maestros agua disponibles para llenar los grupos restantes, se detiene la exploración en esa rama del árbol de búsqueda.

Para que este primer corte sea mas efectivo ordeanmos de mayor a menor los maestros, previo a la llamada de Backtracking, para que corte sea mas efectivo.

Evitar Soluciones Repetidas: Se crea la variable permutacion que representa a los grupos armados hasta ahora, ordenados de mayor a menor, luego verificamos si esa distribución de los grupos ya fue analizada.

Si pertenece a "$ya - analizada$" quiere decir que alguna permutación de los grupos actuales ya ha sido explorada, por lo tanto podemos cortar esa rama.

9.4. Paso 4: Impresión del Resultado

Una vez finalizada la exploración exhaustiva, se imprime el resultado obtenido, que consiste en la asignación de maestros agua a cada grupo.

También se imprime el coeficiente, que es la suma de los cuadrados de los poderes en cada grupo.

10. Código del algoritmo de Backtracking

```
PRIMER_RENGLON = 2

#recibe el nombre del archivo y devuelve una lista de tuplas de (maestro, poder) y una lista de listas vacias (grupos)
def carga(archivo):
    try:
        nombre_archivo = archivo + ".txt"
        with open(nombre_archivo) as archivo:
            lineas = archivo.readlines()
            num_grupos = int(lineas[1].strip())
            grupos = [([], 0)] * num_grupos
            maestros = []
            for linea in lineas[PRIMER_RENGLON:]:
                datos = linea.strip().split(",")
                maestros.append((datos[0], int(datos[1])))
        return grupos, maestros
    except IOError:
        print("Error al abrir el archivo")
        return None

#recibe la lista de maestros y el numero de grupos, devuelve la lista de los grupos resuelta de forma greedy
def sol_Greedy(maestros, num_grupos):
    maestros_ordenados = sorted(maestros, key=lambda x: x[1], reverse=True)
    resultado = [([], 0)] * num_grupos
    for i in range(len(maestros_ordenados)):
        guerrero, poder = maestros_ordenados[i]
        indice_min = min(range(len(resultado)), key=lambda i: resultado[i][1])
        guerreros, sumatoria = resultado[indice_min]
        resultado[indice_min] = (guerreros + [guerrero], sumatoria + poder)
    return resultado


#recibe el indicel maestro actual y verifica que haya suficientes maestros para que ningun grupo quede vacio
def suficientes_guerreros(grupos, actual, total_guerreros):
    vacios = 0
    for i in range(len(grupos)):
        guerreros, _ = grupos[i]
        if len(guerreros) < 1:
            vacios += 1
    if vacios > total_guerreros - actual:
        return False
    return True

#Resuelve el problema por backtracking, sus podas son: si la solucion actual es peor que la mejor, si no se llega a completar los grupos y si una permutacion de la solucion actual ya fue analizada.
def backtracking(maestros, actual, grupos, mejor_result, mejor_suma, cant_grupos, ya_analizado):
    suma_actual = sum(tot * tot for _, tot in grupos)

    if suma_actual >= mejor_suma or not suficientes_guerreros(grupos, actual, len(maestros)):
        return mejor_result, mejor_suma

    permutacion = tuple(sorted(grupo[1] for grupo in grupos))
    if permutacion in ya_analizado:
        return mejor_result, mejor_suma
    ya_analizado.add(permutacion)

    if actual == len(maestros):
        return [grupo for grupo in grupos], suma_actual

    guerrero, poder = maestros[actual]
    for i in range(cant_grupos):
        guerreros, sumatoria = grupos[i]
        grupos[i] = (guerreros + [guerrero], sumatoria + poder)
```

```
62
63          mejor_result, mejor_suma = backtracking(maestros, actual + 1, grupos,
        mejor_result, mejor_suma, cant_grupos, ya_analizado)
64
65          grupos[i] = (guerreros, sumatoria)
66      return mejor_result, mejor_suma
67
68  #recibe el resultado y lo imprime en el formato pedido
69  def impresion_resultado(mejor_result):
70      total = 0
71      for i in range(len(mejor_result)):
72          grupo, suma = mejor_result[i]
73          grupo_str = ', '.join(grupo)
74          print("Grupo " + str(i + 1) + ": " + grupo_str)
75          total += (suma * suma)
76      print("Coeficiente: " + str(total))
77
78
79  def main():
80      archivo = input("Ingrese nombre del archivo:\n")
81      grupos, maestros = carga(archivo)
82
83      result_Greedy = sol_Greedy(maestros, len(grupos))
84      suma_Greedy = sum(tot * tot for _, tot in result_Greedy)
85
86      maestros.sort(key=lambda x: x[1], reverse=True)
87
88      resultado_backtracking, _ = backtracking(maestros, 0, grupos, result_Greedy,
        suma_Greedy, len(grupos), set())
89
90      impresion_resultado(resultado_backtracking)
91
92
93  main()
```

11. Explicación modelo de programación lineal

11.1. Introducción teórica

Variables Continuas (Zmax y Zmin): Representan el poder máximo y mínimo entre los grupos. Hay 2 de estas variables.

Restricciones:

Cada maestro debe estar en exactamente un grupo (n restricciones).

Zmax debe ser mayor o igual al poder de cada grupo (k restricciones).

Zmin debe ser menor o igual al poder de cada grupo (k restricciones).

Los solvers ILP como los utilizados en PULP (por ejemplo, CBC, Gurobi, CPLEX) emplean técnicas como:

Branch-and-bound.

Branch-and-cut.

Heurísticas varias.

Estos algoritmos tienen una complejidad en el peor caso que es exponencial en el número de variables binarias.

Sin embargo, en la práctica, muchas instancias pueden resolverse en tiempos razonables debido a optimizaciones y heurísticas.

Aunque la complejidad teórica es exponencial, en la práctica, los tiempos de ejecución observados pueden ser más manejables debido a las optimizaciones realizadas por los solvers.

11.2. Explicación del algoritmo implementado

11.3. Paso 1: Carga de Datos

El programa comienza solicitando al usuario el nombre del archivo que contiene los datos de entrada. Luego, se lee el archivo y se cargan los datos en la función carga.

Se asume que los datos están en un archivo de texto, donde la primera línea indica el número de grupos a formar y las líneas siguientes contienen los nombres de los maestros agua y sus respectivos poderes.

11.4. Paso 2: Formulación del Problema de Programación Lineal

Una vez cargados los datos, se procede a formular el problema de programación lineal en la función "Grupos-balanceados".

Se utiliza la librería PuLP para definir las variables, restricciones y función objetivo del problema.

Variables de Decisión: Se definen variables binarias (Yi,j) que indican si el maestro agua (i) está asignado al grupo "j", y variables continuas "zj"que representan la suma de los poderes en el grupo "j".

Función Objetivo: Se busca minimizar la diferencia entre el valor máximo Zmax y el valor mínimo Zmin de las sumas de poderes en los grupos.

Restricciones: Se imponen restricciones para garantizar que cada maestro agua sea asignado a exactamente un grupo, que la suma de poderes en cada grupo sea igual a la suma de los poderes de los maestros asignados a ese grupo, y que "Zmax"sea mayor o igual a la suma de poderes en cada grupo, y "Zmin"sea menor o igual a la suma de poderes en cada grupo.

11.5. Paso 3: Resolución del Problema de Programación Lineal

Una vez formulado el problema de programación lineal, se procede a resolverlo utilizando el método solve() de la librería PuLP. Esto devuelve la asignación óptima de maestros agua a los grupos que minimiza la diferencia entre las sumas de los cuadrados de los poderes en cada grupo.

11.6. Paso 4: Impresión del Resultado

Finalmente, se imprime el resultado obtenido, que consiste en la asignación de maestros agua a cada grupo y el coeficiente correspondiente, que es la suma de los cuadrados de los poderes en cada grupo.

12. Código del modelo de programación lineal

```
import pulp

PRIMER_RENGLON = 2

#recibe el nombre del archivo y devuelve una lista de tuplas de (maestro, poder) y una lista de listas vacias (grupos)
def carga(archivo):
    try:
        nombre_archivo = archivo + ".txt"
        with open(nombre_archivo) as archivo:
            lineas = archivo.readlines()
            num_grupos = int(lineas[1].strip())
            grupos = [([], 0) for _ in range(num_grupos)]
            maestros = {}
            for linea in lineas[PRIMER_RENGLON:]:  # O(n)
                datos = linea.strip().split(",")
                maestros[datos[0]] = int(datos[1])
        return grupos, maestros
    except IOError:
        print("Error al abrir el archivo")
        return None

#Resuelve el problema por programacion lineal entera, busca minimizar la resta entre el grupo mas poderoso y el menos poderoso
def Grupos_balanceados(maestros, k):
    nombres = list(maestros.keys())
    poderes = list(maestros.values())

    prob = pulp.LpProblem("Grupos_balanceados", pulp.LpMinimize)

    y = pulp.LpVariable.dicts("y", ((i, j) for i in range(len(nombres)) for j in range(k)), cat='Binary')
    z = pulp.LpVariable.dicts("z", range(k), cat='Continuous')
    Z_max = pulp.LpVariable("Z_max", cat='Continuous')
    Z_min = pulp.LpVariable("Z_min", cat='Continuous')


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

    for j in range(k):
        prob += z[j] == pulp.lpSum(poderes[i] * y[i, j] for i in range(len(nombres)))

    for j in range(k):
        prob += Z_max >= z[j]
        prob += Z_min <= z[j]

    prob += Z_max - Z_min
    prob.solve()

```

```
48      resultado = [([], 0)] * k
49      for i in range(len(nombres)):
50          for j in range(k):
51              if pulp.value(y[i, j]) == 1:
52                  guerreros, sumatoria = resultado[j]
53                  resultado[j] = (guerreros + [nombres[i]], sumatoria + poderes[i])
54      return resultado
55
56  #recibe el resultado y lo imprime en el formato pedido
57  def impresion_resultado(mejor_result):
58      total = 0
59      for i in  range(len(mejor_result)):
60          grupo, suma = mejor_result[i]
61          grupo_str = ', '.join(grupo)
62          print("Grupo " + str(i+1) + ": " + grupo_str)
63          total += (suma*suma)
64      print("Coeficiente: " + str(total))
65
66
67  def main():
68      archivo = input("Ingrese nombre del archivoPL\n")
69      grupos, maestros = carga(archivo)
70      resultado_PL = Grupos_balanceados(maestros, len(grupos))
71      impresion_resultado(resultado_PL)
72
73
74  main()
```

13. Explicación algoritmo de aproximación

Este algoritmo es el pedido por enunciado, el funcionamiento es el siguiente:

Generar los "k"grupos vacíos.

Ordenar de mayor a menor los maestros en función de su habilidad o fortaleza.

Agregar al más habilidoso al grupo con menos habilidad hasta ahora (cuadrado de la suma).

Repetir siguiendo con el siguiente más habilidoso, hasta que no queden más maestros por asignar.

Resumen del flujo del programa:

Entrada: Solicita al usuario el nombre del archivo.

Carga de datos: Lee el archivo, obtiene el número de grupos y los datos de los peleadores.

Procesamiento: Ordena los peleadores por poder y los distribuye entre los grupos para minimizar la diferencia de poder.

Salida: Calcula e imprime la solución aproximada y la suma de los cuadrados de las sumatorias de poder.

Complejidad computacional: La complejidad computacional es O($n * k$).

Donde (n) es la cantidad de maestros y (k) la cantidad de grupos.

14. Código del algoritmo de aproximación

```
PRIMER_RENGLON = 2

#recibe el nombre del archivo y devuelve una lista de tuplas de (maestro, poder) y una lista de listas vacias (grupos)
def carga(archivo):
    try:
        nombre_archivo = archivo + ".txt"
        with open(nombre_archivo) as archivo:
            lineas = archivo.readlines()
            num_grupos = int(lineas[1].strip())
            grupos = [([], 0)] * num_grupos
            maestros = []
            for linea in lineas[PRIMER_RENGLON:]:
                datos = linea.strip().split(",")
                maestros.append((datos[0], int(datos[1])))
        return grupos, maestros
    except IOError:
        print("Error al abrir el archivo")
        return None

#recibe la lista de maestros y el numero de grupos, devuelve la lista de los grupos resuelta de forma greedy
def sol_Greedy(maestros, num_grupos):
    maestros_ordenados = sorted(maestros, key=lambda x: x[1], reverse=True)
    resultado = [([], 0)] * num_grupos
    for i in range(len(maestros_ordenados)):
        guerrero, poder = maestros_ordenados[i]
        indice_min = min(range(len(resultado)), key=lambda i: resultado[i][1])
        guerreros, sumatoria = resultado[indice_min]
        resultado[indice_min] = (guerreros + [guerrero], sumatoria + poder)
    return resultado

#recibe el resultado y lo imprime en el formato pedido
def impresion_resultado(mejor_result):
    total = 0
    for i in range(len(mejor_result)):
        grupo, suma = mejor_result[i]
        grupo_str = ', '.join(grupo)
        print("Grupo " + str(i + 1) + ": " + grupo_str)
        total += (suma * suma)
    print("Coeficiente: " + str(total))


def main():
    archivo = input("Ingrese nombre del archivo:\n")
    grupos, maestros = carga(archivo)

    result_Greedy = sol_Greedy(maestros, len(grupos))

    impresion_resultado(result_Greedy)


main()
```

15. Explicación algoritmo de aproximación Greedy

A continuación se pasa a detallar el paso a paso del funcionamiento del algoritmo.

15.1. Paso 1: Carga de Datos

El programa comienza solicitando al usuario el nombre del archivo que contiene los datos de entrada. Luego, se leen los datos en la función carga. Se asume que los datos están en un archivo de texto, donde la primera línea contiene el número de grupos a formar y las líneas siguientes contienen los nombres de los maestros agua y sus respectivos poderes.

15.2. Paso 2: Solución Greedy

Una vez cargados los datos, se procede a resolver el problema mediante un enfoque greedy en la función "sol-Greedy2". Este enfoque consiste en ordenar a los maestros agua de mayor a menor poder y asignarlos a los grupos en un patrón de ida y vuelta (zigzag) para intentar equilibrar los poderes entre los grupos.

15.3. Paso 3: Impresión del Resultado

Finalmente, se imprime el resultado obtenido, que consiste en la asignación de maestros agua a cada grupo y el coeficiente correspondiente, que es la suma de los cuadrados de los poderes en cada grupo.

15.4. Complejidad computacional:

La complejidad computacional es O($nlogn$).

Donde (n) es la cantidad de maestros.

15.5. Código del algoritmo de aproximación Greedy

```
PRIMER_RENGLON = 2

#recibe el nombre del archivo y devuelve una lista de tuplas de (maestro, poder) y una lista de listas vacias (grupos)
def carga(archivo):
    try:
        nombre_archivo = archivo + ".txt"
        with open(nombre_archivo) as archivo:
            lineas = archivo.readlines()
            num_grupos = int(lineas[1].strip())
            grupos = [([], 0)] * num_grupos
            maestros = []
            for linea in lineas[PRIMER_RENGLON:]:
                datos = linea.strip().split(",")
                maestros.append((datos[0], int(datos[1])))
        return grupos, maestros
    except IOError:
        print("Error al abrir el archivo")
        return None

#recibe la lista de maestros y el numero de grupos, devuelve la lista de los grupos resuelta de forma greedy
def sol_Greedy2(maestros, num_grupos):
    maestros_ordenados2 = sorted(maestros, key=lambda x: x[1], reverse=True)
    resultado2 = [([], 0)] * num_grupos
    ida = True
    grupo = 0
```

```
    for guerrero in maestros_ordenados2:
        guerreros, sumatoria = resultado2[grupo]
        resultado2[grupo] = (guerreros + [guerrero[0]], sumatoria + guerrero[1])
        if ida:
            grupo += 1
            if grupo == num_grupos:
                grupo = num_grupos - 1
                ida = False
        else:
            grupo -= 1
            if grupo == -1:
                grupo = 0
                ida = True
    return resultado2

#recibe el resultado y lo imprime en el formato pedido
def impresion_resultado(mejor_result):
    total = 0
    for i in range(len(mejor_result)):
        grupo, suma = mejor_result[i]
        grupo_str = ', '.join(grupo)
        print("Grupo " + str(i + 1) + ": " + grupo_str)
        total += (suma * suma)
    print("Coeficiente: " + str(total))


def main():
    archivo = input("Ingrese nombre del archivo:\n")
    grupos, maestros = carga(archivo)

    result_Greedy2 = sol_Greedy2(maestros, len(grupos))

    impresion_resultado(result_Greedy2)


main()
```

16. Testeo del algoritmo

El algoritmo fue probado tanto con las pruebas brindadas por la cátedra, como por las propias creadas y siempre cumplió con la condición pedida para el valor óptimo.

Las pruebas propias generadas tuvieron 2 ramas.

La primer rama, son pruebas de valores puntuales, donde se buscaba probar a los algoritmos implementados bajo ciertas condiciones de entrada, siendo esto también muy útil para comparar resultados del .Algoritmo de aproximaciónrespecto al valor óptimo en diversas condiciones.

La segunda rama son pruebas con valores de entrada muy grande, que solo son utilizados por el algoritmo de aproximación. Esto se hace para demostrar la utilidad del mismo, ya que si bien su respuesta no es óptima, sirve mucho para estos casos de volumenes de datos muy grandes.

16.1. Las pruebas de la primer rama son:

16.2. MasGrandesQuePequeños:

5

Siku, 23

Hasook, 34

Desna, 11

Desna I, 350

Siku I, 375

Unalaq, 452

Sura, 483

Misu, 394

Katara, 404

Eska, 563

Resultado esperado:

Grupo 1: Misu, Siku I.

Grupo 2: Katara, Desna I.

Grupo 3: Unalaq, Hasook, Desna.

Grupo 4: Sura, Siku.

Grupo 5: Eska.

Coeficiente: 1979891.

16.3. MasPequeñosQueGrandes:

5

Siku, 500

Hasook, 600

Desna, 400

Desna I, 35

Siku I, 37

Unalaq, 12

Sura, 13

Misu, 22

Katara, 27

Eska, 11

Resultado esperado:

Grupo 1: Hasook.

Grupo 2: Siku.

Grupo 3: Desna.

Grupo 4: Siku I, Katara, Sura.

Grupo 5: Desna I, Misu, Unalaq, Eska.

Coeficiente: 782329.

16.4. Poderes-Grandes:

5

Siku, 500

Hasook, 600

Desna, 400

Desna I, 350

Siku I, 375

Unalaq, 452

Sura, 483

Misu, 394

Katara, 404

Eska, 563

Resultado esperado:

Grupo 1: Unalaq, Katara.

Grupo 2: Sura, Desna.

Grupo 3: Siku, Misu.

Grupo 4: Eska, Siku I.

Grupo 5: Hasook, Desna I.

Coeficiente: 4094005.

16.5. Poderes-Pequeños:

5

Siku, 20

Hasook, 30

Desna, 11

Desna I, 35

Siku I, 37

Unalaq, 12

Sura, 27

Misu, 31

Katara, 29

Eska, 13

Resultado esperado:

Grupo 1: Katara, Sura.

Grupo 2: Hasook, Siku.

Grupo 3: Misu, Eska.

Grupo 4: Desna I, Unalaq.

Grupo 5: Siku I, Desna.

Coeficiente: 12085.

16.6. Las pruebas de la segunda rama son:

16.7. 120-8

8

Ming-Hua, 153

Yakone, 792

Tho, 756

Rafa, 576

Unalaq, 874

Wei, 934

Katara, 121

Hama, 637

Sura, 239

Pakku, 365

La, 30

Amon, 455

Ming-Hua I, 451

Amon I, 316

Tonraq, 786

Sangok, 193

Kuruk, 893

Hama I, 57

Sura I, 153

Misu, 775

Ming-Hua, 153

Yakone, 792

Tho, 756

Rafa, 576

Unalaq, 874

Wei, 934

Katara, 121

Hama, 637
Sura, 239
Pakku, 365
La, 30
Amon, 455
Ming-Hua I, 451
Amon I, 316
Tonraq, 786
Sangok, 193
Kuruk, 893
Hama I, 57
Sura I, 153
Misu, 775
Ming-Hua, 153
Yakone, 792
Tho, 756
Rafa, 576
Unalaq, 874
Wei, 934
Katara, 121
Hama, 637
Sura, 239
Pakku, 365
La, 30
Amon, 455
Ming-Hua I, 451
Amon I, 316
Tonraq, 786
Sangok, 193
Kuruk, 893
Hama I, 57
Sura I, 153
Misu, 775
Ming-Hua, 153
Yakone, 792
Tho, 756
Rafa, 576
Unalaq, 874
Wei, 934
Katara, 121
Hama, 637

Sura, 239
Pakku, 365
La, 30
Amon, 455
Ming-Hua I, 451
Amon I, 316
Tonraq, 786
Sangok, 193
Kuruk, 893
Hama I, 57
Sura I, 153
Misu, 775
Ming-Hua, 153
Yakone, 792
Tho, 756
Rafa, 576
Unalaq, 874
Wei, 934
Katara, 121
Hama, 637
Sura, 239
Pakku, 365
La, 30
Amon, 455
Ming-Hua I, 451
Amon I, 316
Tonraq, 786
Sangok, 193
Kuruk, 893
Hama I, 57
Sura I, 153
Misu, 775
Ming-Hua, 153
Yakone, 792
Tho, 756
Rafa, 576
Unalaq, 874
Wei, 934
Katara, 121
Hama, 637
Sura, 239

Pakku, 365

La, 30

Amon, 455

Ming-Hua I, 451

Amon I, 316

Tonraq, 786

Sangok, 193

Kuruk, 893

Hama I, 57

Sura I, 153

Misu, 775

Resultado esperado:

Grupo 1: 'Wei', 'Kuruk', 'Yakone', 'Tonraq', 'Tho', 'Tho', 'Amon', 'Amon', 'Pakku', 'Amon I', 'Sura', 'Ming-Hua', 'Ming-Hua', 'Hama I', 'Hama I'.

Grupo 2: 'Wei', 'Kuruk', 'Yakone', 'Tonraq', 'Tho', 'Tho', 'Amon', 'Amon', 'Pakku', 'Amon I', 'Sura', 'Sura I', 'Sura I', 'Hama I', 'Hama I'.

Grupo 3: 'Wei', 'Unalaq', 'Yakone', 'Tonraq', 'Misu', 'Hama', 'Rafa', 'Amon', 'Pakku', 'Amon I', 'Sura', 'Ming-Hua', 'Katara', 'Katara', 'La'.

Grupo 4: 'Wei', 'Unalaq', 'Yakone', 'Tonraq', 'Misu', 'Hama', 'Rafa', 'Amon', 'Pakku', 'Amon I', 'Sura', 'Sura I', 'Katara', 'Katara', 'La'.

Grupo 5: 'Wei', 'Unalaq', 'Yakone', 'Tonraq', 'Misu', 'Hama', 'Rafa', 'Ming-Hua I', 'Ming-Hua I', 'Sura', 'Sangok', 'Sangok', 'Ming-Hua', 'Hama I', 'La', 'La'.

Grupo 6: 'Wei', 'Unalaq', 'Yakone', 'Tonraq', 'Misu', 'Hama', 'Rafa', 'Ming-Hua I', 'Ming-Hua I', 'Sura', 'Sangok', 'Sangok', 'Sura I', 'Hama I', 'La', 'La'.

Grupo 7: 'Kuruk', 'Kuruk', 'Unalaq', 'Misu', 'Tho', 'Hama', 'Rafa', 'Ming-Hua I', 'Pakku', 'Amon I', 'Sangok', 'Ming-Hua', 'Ming-Hua', 'Katara'.

Grupo 8: 'Kuruk', 'Kuruk', 'Unalaq', 'Misu', 'Tho', 'Hama', 'Rafa', 'Ming-Hua I', 'Pakku', 'Amon I', 'Sangok', 'Sura I', 'Sura I', 'Katara'.

Coeficiente: 410927484.

16.8. 60-5:

5

Siku, 169

Hasook, 27

Desna, 76

Desna I, 156

Siku I, 196

Unalaq, 132

Sura, 102

Misu, 198

Katara, 200

Eska, 74

Siku, 169

Hasook, 27
Desna, 76
Desna I, 156
Siku I, 196
Unalaq, 132
Sura, 102
Misu, 198
Katara, 200
Eska, 74
Siku, 169
Hasook, 27
Desna, 76
Desna I, 156
Siku I, 196
Unalaq, 132
Sura, 102
Misu, 198
Katara, 200
Eska, 74
Siku, 169
Hasook, 27
Desna, 76
Desna I, 156
Siku I, 196
Unalaq, 132
Sura, 102
Misu, 198
Katara, 200
Eska, 74
Siku, 169
Hasook, 27
Desna, 76
Desna I, 156
Siku I, 196
Unalaq, 132
Sura, 102
Misu, 198
Katara, 200
Eska, 74
Siku, 169
Hasook, 27

Desna, 76

Desna I, 156

Siku I, 196

Unalaq, 132

Sura, 102

Misu, 198

Katara, 200

Eska, 74

Resultado esperado:

Grupo 1: 'Katara', 'Katara', 'Siku I', 'Siku I', 'Desna I', 'Desna I', 'Unalaq', 'Sura', 'Sura', 'Eska', 'Hasook', 'Hasook', 'Hasook'.

Grupo 2: 'Katara', 'Misu', 'Misu', 'Siku', 'Siku', 'Desna I', 'Unalaq', 'Unalaq', 'Desna', 'Eska', 'Eska'.

Grupo 3: 'Katara', 'Misu', 'Misu', 'Siku', 'Siku', 'Desna I', 'Unalaq', 'Sura', 'Sura', 'Desna', 'Eska', 'Hasook'.

Grupo 4: 'Katara', 'Misu', 'Siku I', 'Siku I', 'Siku', 'Desna I', 'Unalaq', 'Sura', 'Desna', 'Desna', 'Eska', 'Hasook'.

Grupo 5: 'Katara', 'Misu', 'Siku I', 'Siku I', 'Siku', 'Desna I', 'Unalaq', 'Sura', 'Desna', 'Desna', 'Eska', 'Hasook'.

Coeficiente: 12736526.

16.9. 90-6:

6

Senna, 336

Senna I, 66

Senna II, 856

Huu, 580

Huu I, 154

Amon, 841

La, 669

Hasook, 130

Sura, 56

Wei, 189

Hasook I, 811

Rafa, 247

Ming-Hua, 305

Tonraq, 267

Pakku, 678

Senna, 336

Senna I, 66

Senna II, 856

Huu, 580

Huu I, 154
Amon, 841
La, 669
Hasook, 130
Sura, 56
Wei, 189
Hasook I, 811
Rafa, 247
Ming-Hua, 305
Tonraq, 267
Pakku, 678
Senna, 336
Senna I, 66
Senna II, 856
Huu, 580
Huu I, 154
Amon, 841
La, 669
Hasook, 130
Sura, 56
Wei, 189
Hasook I, 811
Rafa, 247
Ming-Hua, 305
Tonraq, 267
Pakku, 678
Senna, 336
Senna I, 66
Senna II, 856
Huu, 580
Huu I, 154
Amon, 841
La, 669
Hasook, 130
Sura, 56
Wei, 189
Hasook I, 811
Rafa, 247
Ming-Hua, 305
Tonraq, 267
Pakku, 678

Senna, 336

Senna I, 66

Senna II, 856

Huu, 580

Huu I, 154

Amon, 841

La, 669

Hasook, 130

Sura, 56

Wei, 189

Hasook I, 811

Rafa, 247

Ming-Hua, 305

Tonraq, 267

Pakku, 678

Senna, 336

Senna I, 66

Senna II, 856

Huu, 580

Huu I, 154

Amon, 841

La, 669

Hasook, 130

Sura, 56

Wei, 189

Hasook I, 811

Rafa, 247

Ming-Hua, 305

Tonraq, 267

Pakku, 678

Resultado esperado:

Grupo 1: 'Senna II', 'Amon', 'Hasook I', 'Pakku', 'La', 'Huu', 'Senna', 'Ming-Hua', 'Tonraq', 'Rafa', 'Wei', 'Huu I', 'Hasook', 'Senna I', 'Sura'.

Grupo 2: 'Senna II', 'Amon', 'Hasook I', 'Pakku', 'La', 'Huu', 'Senna', 'Ming-Hua', 'Tonraq', 'Rafa', 'Wei', 'Huu I', 'Hasook', 'Senna I', 'Sura'.

Grupo 3: 'Senna II', 'Amon', 'Hasook I', 'Pakku', 'La', 'Huu', 'Senna', 'Ming-Hua', 'Tonraq', 'Rafa', 'Wei', 'Huu I', 'Hasook', 'Senna I', 'Sura'.

Grupo 4: 'Senna II', 'Amon', 'Hasook I', 'Pakku', 'La', 'Huu', 'Senna', 'Ming-Hua', 'Tonraq', 'Rafa', 'Wei', 'Huu I', 'Hasook', 'Senna I', 'Sura'.

Grupo 5: 'Senna II', 'Amon', 'Hasook I', 'Pakku', 'La', 'Huu', 'Senna', 'Ming-Hua', 'Tonraq', 'Rafa', 'Wei', 'Huu I', 'Hasook', 'Senna I', 'Sura'.

Grupo 6: 'Senna II', 'Amon', 'Hasook I', 'Pakku', 'La', 'Huu', 'Senna', 'Ming-Hua', 'Tonraq', 'Rafa', 'Wei', 'Huu I', 'Hasook', 'Senna I', 'Sura'.

Coeficiente: 229525350.

17. Cota empírica de aproximación

Para determinar que tan buena aproximación da nuestro algoritmo vamos a listar los resultados obtenidos en todas las pruebas de la cátedra con el algoritmo de "Backtracking"(óptimo) y el de ."Aproximación"(aproximado).

17.1. Resultados óptimos

5-2.txt

Coeficiente: 1894340.

6-3.txt

Coeficiente: 1640690.

6-4.txt

Coeficiente: 807418.

8-3.txt

Coeficiente: 4298131.

10-3.txt

Coeficiente: 385249.

10-5.txt

Coeficiente: 355882.

10-10.txt

Coeficiente: 172295.

11-5.txt

Coeficiente: 2906564

14-3.txt

Coeficiente: 15659106

14-4.txt

Coeficiente: 15292055

14-6.txt

Coeficiente: 10694510

15-4.txt

Coeficiente: 4311889

15-6.txt

Coeficiente: 6377225

17-5.txt

Coeficiente: 15974095

17-7.txt

Coeficiente: 11513230

17-10.txt

Coeficiente: 5427764

18-6.txt

Coeficiente: 10322822

18-8.txt

Coeficiente: 11971097

20-4.txt

Coeficiente: 21081875

20-5.txt

Coeficiente: 16828799

20-8.txt

Coeficiente: 11423730

17.2. Resultados aproximados

5-2.txt

Coeficiente: 1894340.

6-3.txt

Coeficiente: 1640690.

6-4.txt

Coeficiente: 807418.

8-3.txt

Coeficiente: 4298131.

10-3.txt

Coeficiente: 385249.

10-5.txt

Coeficiente: 355882.

10-10.txt

Coeficiente: 172295.

11-5.txt

Coeficiente: 2906564.

14-3.txt

Coeficiente: 15664276.

14-4.txt

Coeficiente: 15292085.

14-6.txt

Coeficiente: 10700172.

15-4.txt

Coeficiente: 4317075.

15-6.txt

Coeficiente: 6377501.

17-5.txt

Coeficiente: 15975947.

17-7.txt

Coeficiente: 11513230.

17-10.txt

Coeficiente: 5430512.

18-6.txt

Coeficiente: 10325588.

18-8.txt

Coeficiente: 12000279.

20-4.txt

Coeficiente: 21083935.

20-5.txt

Coeficiente: 16838539.

20-8.txt

Coeficiente: 11423826.

17.3. Interpretación de los resultados

Con todos los resultados podemos ver que en muchos casos los resultados coinciden y casos en donde no, vamos a analizarlos todos:

Resultados coinciden: 5-2; 6-3; 6-4; 8-3; 10-3; 10-5; 10-10; 11-5; 17-7.

Resultados no coinciden: 14-3; 14-4; 14-6; 15-4; 15-6; 17-5; 17-10; 18-6; 18-8; 20-4; 20-5; 20-8.

Con esto podemos ver una particularidad, y es que para las pruebas con pocos maestros es mas posible que el algoritmo de aproximación encuentre resultados óptimos.

Mientras que con el aumento de los valores, esta posibilidad va decreciendo.

Por lo que, podemos estimar que las soluciones que encontramos a los valores de entrada muy grandes no van a ser óptimos.

Pero, la pregunta es que tan aproximados pueden ser, y eso lo vamos a ver con los casos de no concidencia.

Para eso, vamos a calcular la diferencia entre los óptimos y las aproximaciones de los diferentes.

17.4. Cálculo de diferencias de aproximación

14-3 = 5170 (0.03 %).

14-4 = 30 (0.0001 %).

14-6 = 5662 (0.05 %).

15-4 = 5176 (0.1 %).

15-6 = 276 (0.004 %).

17-5 = 1852 (0.01 %).

17-10 = 2748 (0.05 %).

18-6 = 2766 (0.02 %).

18-8 = 29182 (0.2 %).

20-4 = 2060 (0.009 %).

20-5 = 9740 (0.05 %).

20-8 = 96 (0.00008 %).

Con esto, podemos llegar a 2 valores:

Error máximo de aproximación: 0.2 %.

Error promedio de aproximación: 0.0153 %.

Para el error máximo se tomo solo el mayor porcentaje de error.

Para el error promedio se tomo todos los valores porcentuales obtenidos (incluso los que dan igual, es decir "0 %"), se sumaron entre sí y se dividieron por la cantidad de pruebas.

Pero es importante recordar que no está demostrado de forma teórica estos valores, sino de forma empírica con las pruebas que tenemos.

18. Explicación del programa de gráficos

Se utilizó la biblioteca "matplotlib"para realizar unos gráficos que nos muestren las curvas de tendencia de complejidad computacional que siguen las pruebas realizadas.

En este caso, además de las curvas de tendencia de cada prueba en particular (color azul), se trazó una curva punteada referencial (color rojo), para poder corroborar si efectivamente la complejidad computacional estudiada responde a la detectada con la ejecución del programa.

19. Código del algoritmo de los gráficos

```
import time
import math
import matplotlib.pyplot as plt
from funciones import carga, aproximacion1, aproximacion2, backtracking

FACTOR=1

def medir_tiempo(funcion, maestros, num_grupos,grupos,result_Greedy,suma_Greedy):
    inicio = time.time()
    if funcion.__name__ == 'backtracking':
        resultado, suma = backtracking(maestros, 0, grupos, result_Greedy,
    suma_Greedy, num_grupos, set())
    else:
        resultado = funcion(maestros, num_grupos)
    fin = time.time()
    return fin - inicio


archivo = input("Ingrese nombre del archivo: ")
grupos, maestros = carga(archivo)

num_grupos = len(grupos)
n = len(maestros)
tiempos_aprox1 = []
tiempos_aprox2 = []
tiempos_backtracking = []

for i in range(1, n + 1):
  result_aprox1 = aproximacion1(maestros[:i], len(grupos))
  suma_aprox1 = sum(tot * tot for _, tot in result_aprox1)

  #tiempo_aprox1 = medir_tiempo(aproximacion1, maestros[:i], num_grupos,grupos,0,0)
  #tiempo_aprox2 = medir_tiempo(aproximacion2, maestros[:i], num_grupos,grupos,0,0)
  tiempo_backtracking = medir_tiempo(backtracking, maestros[:i], num_grupos,grupos,
    result_aprox1,suma_aprox1)

  #tiempos_aprox1.append(tiempo_aprox1)
  #tiempos_aprox2.append(tiempo_aprox2)
  tiempos_backtracking.append(tiempo_backtracking)

#Graficar los tiempos de ejecuci n
escala = 10**6 * FACTOR
tiempos_aprox1_escalados = [t * escala for t in tiempos_aprox1]
tiempos_aprox2_escalados = [t * escala for t in tiempos_aprox2]
tiempos_backtracking_escalados = [t * escala for t in tiempos_backtracking]

# Complejidad te rica para comparaci n
dos_a_la_n = [2 ** i for i in range(1, n + 1)]
#complejidad_aproximacion1 = [(i*num_grupos + i * math.log(i)) for i in range(1, n
    + 1)]
#complejidad_aproximacion2 = [(i * math.log(i)) for i in range(1, n + 1)]


plt.figure(figsize=(12, 8))
#plt.plot(range(1, n + 1), tiempos_aprox1_escalados, marker='o', label='sol_aprox1
    ')
#plt.plot(range(1, n + 1), tiempos_aprox2_escalados, marker='x', label='sol_Greedy2
    ')
plt.plot(range(1, n + 1), tiempos_backtracking_escalados, marker='s', label='
    backtracking')
#plt.plot(range(1, n + 1), complejidad_aproximacion1, linestyle='--', label='O(n*k
    + n * log(n))')
#plt.plot(range(1, n + 1), complejidad_aproximacion2, linestyle='--', label='O(n *
    log(n))')
plt.plot(range(1, n + 1), dos_a_la_n, linestyle='--', label='O(2^n)')

plt.xlabel('N mero de maestros')
plt.ylabel('Tiempo de ejecuci n (escalado)')
```

```
61 plt.title('Tiempo de ejecuci n de los algoritmos')
62 plt.legend()
63 plt.grid(True)
64 plt.show()
```

20. Explicación de gráficos

A continuación se pasa a exhibir los gráficos obtenidos con el programa antes mencionado, para luego hacer el análisis práctico de la complejidad computacional y ver si los datos se corroboran con lo analizado previamente.

21. Gráficos

21.1. Backtracking

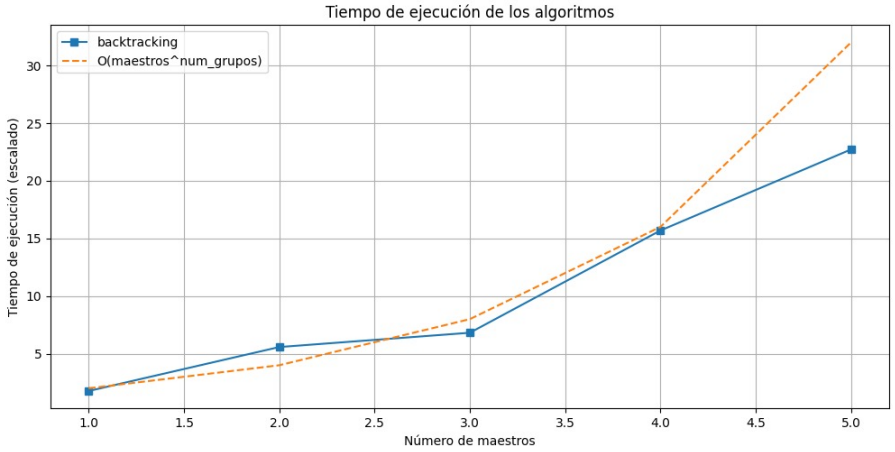

Figura 1: Gráfico de prueba de tiempo de ejecución de 5 maestros.

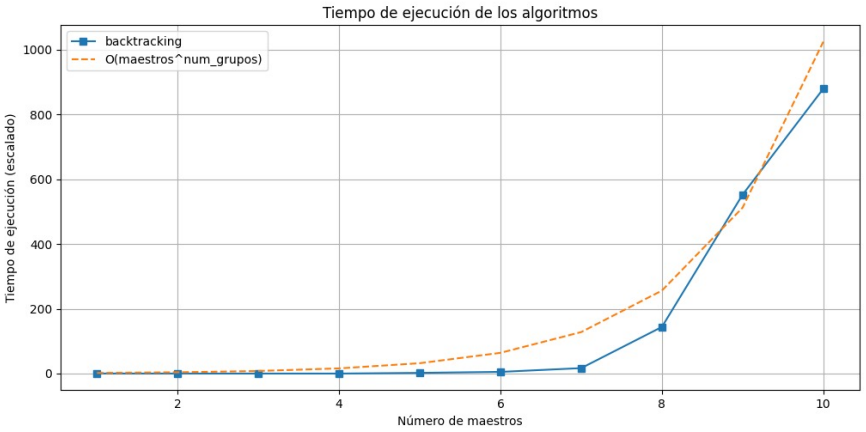

Figura 2: Gráfico de prueba de tiempo de ejecución de 10 maestros (a).

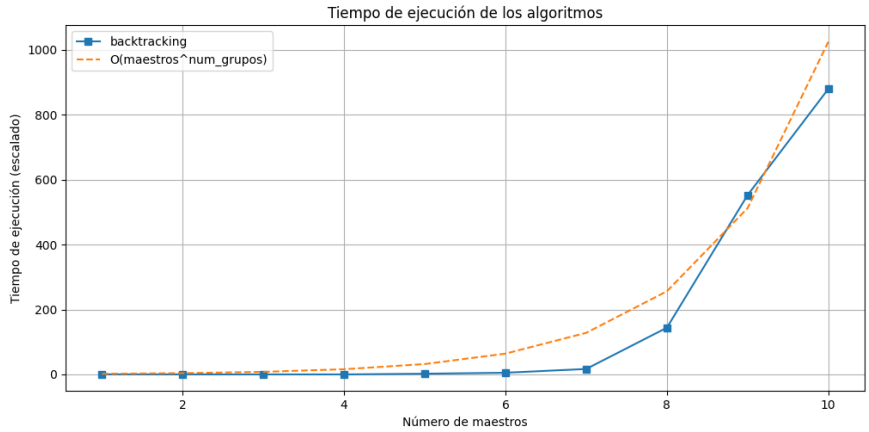

Figura 3: Gráfico de prueba de tiempo de ejecución de 10 maestros (b).

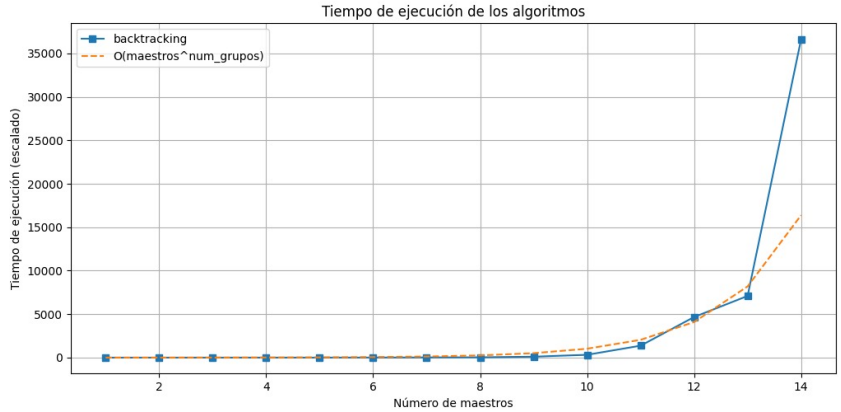

Figura 4: Gráfico de prueba de tiempo de ejecución de 14 maestros.

21.2. Aproximación (Obligatoria)

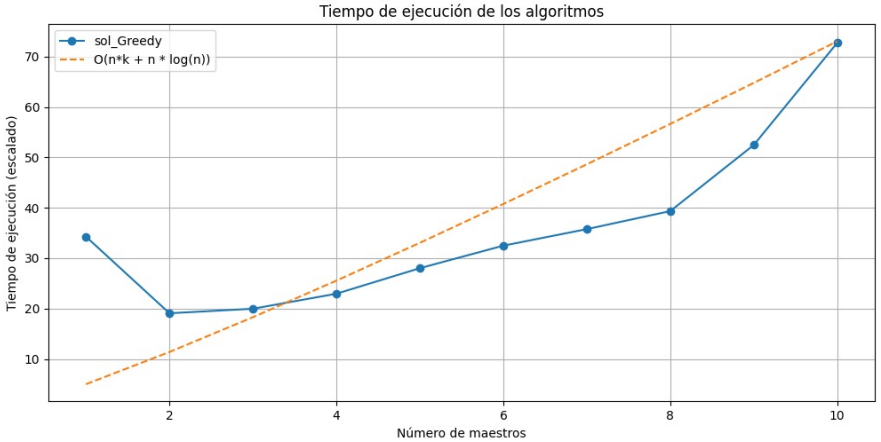

Figura 5: Gráfico de prueba de tiempo de ejecución de 10 maestros.

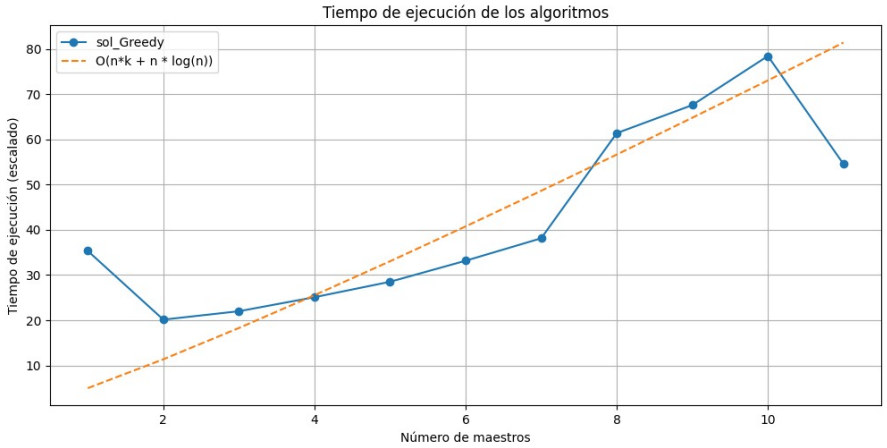

Figura 6: Gráfico de prueba de tiempo de ejecución de 14 maestros.

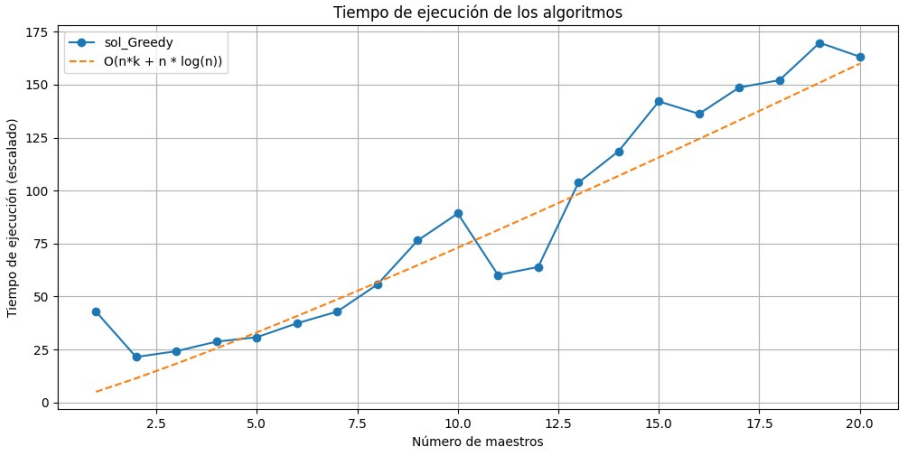

Figura 7: Gráfico de prueba de tiempo de ejecución de 20 maestros.

21.3. Aproximación (Greedy)

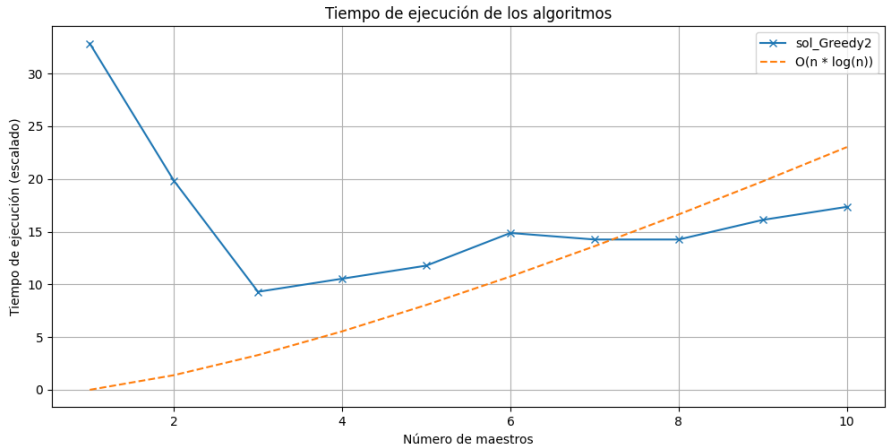

Figura 8: Gráfico de prueba de tiempo de ejecución de 10 maestros.

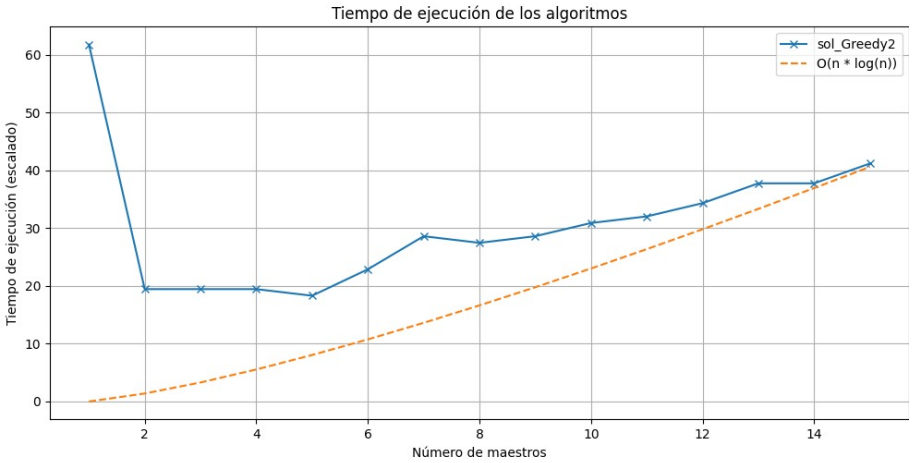

Figura 9: Gráfico de prueba de tiempo de ejecución de 14 maestros.

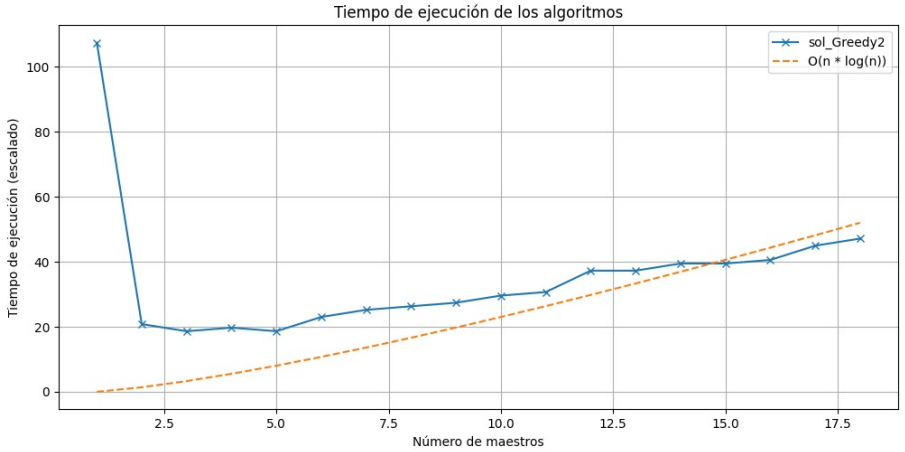

Figura 10: Gráfico de prueba de tiempo de ejecución de 17 maestros.

22. Interpretación de los gráficos

Lo primero que podemos observar con los gráficos obtenidos y las curvas teóricas, es la gran diferencia de complejidad computacional entre los algoritmos que habiamos analizado previamente.

Siendo la mayor diferencia entre los gráficos de "Backtrackingz los de .ªproximación".

Detectando asi 2 conclusiones interesantes:

Primero: Es vital tener el algoritmo de "Backtracking"para llegar a los resultados óptimos del problema siempre y cuando se pueda manejar el volumen de entrada.

Segundo: Es de gran utilidad contar con buenos algoritmos de aproximación para poder manejar grandes volumenes de datos y obtener resultados no muy alejados de la realidad.

Luego de ver todos los gráficos en profundidad, podemos sacar varias conclusiones:

Apreciaciones de los gráficos:

- **Correlación con la curva teórica**: Como podemos apreciar en todos los gráficos la curva teórica aproxima muy bien los valores que toma el programa con su funcionamiento.
- **Continuidad de la curva**: La curva se mantiene pareja y con un crecimiento constante de forma bastante armónica.
- **¿Es fiel a nuestro análisis teórico?**: Una vez obtenidos los gráficos podemos afirmar que "SI", si bien los gráficos tienen sus pequeñas variaciones, se ve que la tendencia de la curva calculada teóricamente es respetada por lo que podemos afirmar que tanto de manera teórica como práctica la Çomplejidad Computacional Temporal"de los algoritmos implementados.

Conclusión

23. Conclusiones

Al final del trabajo se consiguió completar todos los requisitos del enunciado, y se pudo obtener resultados exitosos de las hipótesis y propuestas que se fueron desarrollando a lo largo del trabajo.

Algunas de las ideas principales que se lograron completar son:

- Se llegó a la resolución del problema mediante a un "Algoritmo Backtracking" óptimo.
- Se llegó a la resolución del problema mediante a un "Modelo de Programación Lineal".
- Se llegó a una resolución aproximada del problema mediante a un "Algoritmo de Aproximación".
- Se llegó a una resolución aproximada del problema mediante a un "Algoritmo de Aproximación Greedy".
- Se realizó código e implementación total de los algoritmos pedidos en formato funcional.
- - Se verificó mediante a gráficos la complejidad teórica, y la tendencia del gráfico a respetar el original de una función.
- - Se midió de forma real los tiempos de ejecución de los distintos algoritmos.

Para concluir, se cumplió con todas las expectativas iniciales y se pudo resolver el problema de forma óptima y aproximada.