Teoría de Algoritmos
(75.29) Curso Echevarría

Trabajo práctico 1
Problemas NP-Completos para la defensa de la Tribu del Agua

2 de junio de 2024

Isidro Hector Borthaburu
108901
iborthaburu@fi.uba.ar

María Delfina Cano Ros Langrehr
109338
mcano@fi.uba.ar

Delfina Maria Fuhr
111119
dfuhr@fi.uba.ar

Índice

1. Introducción

Durante este informe utilizaremos el problema de la tribu del agua o también como nosotros lo decidimos llamar .[el] problema de los maestros". Este consiste en que se busca separar a todos los maestros de la tribu de agua para que defiendan su pueblo de los ataques. Para hacer esto deben dividirse en subgrupos para así generar un ataque constante y fuerte, por eso se debe decidir como dividir los maestros de manera más estratégica ya que cada uno cuenta con un poder de habilidad/fortaleza (número real positivo). Para esto se decide también la cantidad de subgrupos que debe haber para poder defender la tribu.

Comenzando, probaremos que este problema se encuentra en NP y que también está en NP-Completo utilizando una reducción. Además, implementaremos un algoritmo por *backtracking* que de la solución óptima al problema planteado, además incluiremos mediciones de tiempo entre dos soluciones óptima pero una utilizando un algoritmo greedy como aproximación que convierte al algoritmo en más óptimo. Continuando, planteáremos un modelo de programación lineal y obtendremos una aproximación muy cercana al resultado óptimo y luego incluiremos su código. Por último realizamos otro algoritmo de aproximación, demostraremos su complejidad, compararemos con un algoritmo exacto, tomaremos mediciones de tiempo y más.

2. Demostración que el Problema de la Tribu del Agua es NP

En primer lugar, vamos a demostrar que el problema de la tribu del agua es un problema que se encuentra en NP. Para poder realizar esto, debemos realizar un validador que tenga complejidad polinomial y si esto es válido, se puede decir que el problema se encuentra en NP.

Planteamos el validador donde en este recibimos por parámetro

- el conjunto S de todos los subgrupos $S = [S_1, S_2, ..., S_k]$
- el valor k que es igual a la cantidad de subgrupos que debe haber
- el valor B que debe ser igual o menor a la suma total que se verificará que de como resultado la *ecuación 1*
- el conjunto de los maestros del agua x_j donde j va de 1 a n (cantidad total de maestros), para representar este conjunto de maestros utilizamos la letra L. Cabe mencionar que cada maestro tiene un valor positivo que representa su habilidad/fortaleza al combatir, y este valor es el que se utiliza para calcular la ecuación 1, por lo tanto, L es un vector de números.

La suma total a calcular es:

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

Lo que se busca validar en la función validadora es que:

- la cantidad de subconjuntos s de S es k
- que todos los maestros de cada subconjunto s estén en L
- que todos los maestros estén únicamente en un grupo y en ese grupo una sola vez
- que todos los maestros estén en un subgrupo
- que el resulatado de la ecuación sea menor o igual a B
- que la cantidad total de soldados en L sea mayor o igual a k, sino no es posible que haya k cantidad de subgrupos con algún maestro

```python
def validador(S, k, B, L):
    if len(S) != k:
        return False

    if len(L) > k:
        return False

    vector_maestros = []
    suma_total = 0

    for s in S:
        suma_subgrupo = 0
        for soldado in s:
            if soldado not in L:
                return False
            if soldado not in vector_maestros:
                vector_maestros.add(soldado)
            else:
                return False
            suma_subgrupo += soldado
        suma_total += (suma_subgrupo **2        )

    if len(vector_maestros) != len(L):
        return False

    if suma_total > B:
        return False

    return True
```

Por lo tanto, la complejidad algorítmica de nuestra función es $O(k * s)$ donde k es la cantidad de subgrupos en que se divide L y s es la cantidad de maestros en cada subgrupo.

De esta forma, demostramos que el problema de la tribu del agua se encuentra en NP ya que la complejidad algorítmica del validador es polinomial.

3. Demostración que el problema de los maestros del agua es NP-Completo

En esta sección demostraremos que el problema de los maestros del agua es NP-Completo (NP-C), para poder realizar esto debemos buscar un problema NP-C más “sencillo"que este, lo denominamos *problema X*. Por lo tanto, debemos reducir este problema X al problema de los maestros y poder resolver X usando la “caja negra"que resuelve el problema de los maestros. Cuando mencionamos la “caja negra"nos referimos a un problema de decisión donde la “caja"devuelve los valores booleanos true o false. Entonces al mencionar esta caja lo que buscamos es que el problema de decisión sea si existe una partición de k subgrupos, tal que la suma total calculada en la *ecuación 1* sea igual a B.

El problema que decidimos reducir al problema de los maestros fue “El problema de los libros y las cajas", consideramos usar los problemas *k-coloreo* o *k-partition*, pero el que nos convenció fue este. Este problema fue tomado en el primer parcial del segundo cuatrimestre del 2022 y también fue dado en la clase de repaso como “el problema de las bolsas” (también se encuentra en RPL) en esa cursada también, ambos problemas son iguales simplemente que en uno se guardan elementos con pesos en bolsas y en este se guardan libros en cajas. En el vídeo resolviendo ese respectivo parcial, se aclara que este problema es **NP-Completo**.

Este es el enunciado del problema de las cajas:

“Tenés una colección de n libros con diferentes espesores, que pueden estar entre 1 y n (valores no necesariamente enteros). Tu objetivo es guardar esos libros en la menor cantidad de cajas. Todas las cajas que disponés son de la misma capacidad P (se asegura que $P \geq n$). No se puede partir un libro para que vaya en múltiples cajas, pero sí podés poner múltiples libros en una misma caja, siempre y cuando no superen esa capacidad P”. Como se menciono, el objetivo de este problema

es minimizar la cantidad de cajas a usar.

Por lo tanto, hacemos la siguiente reducción:

$$PC \leq_p PM \tag{2}$$

Es decir, el problema de las cajas se reduce polinomialmente al problema de los maestros. Como PM es al menos tan difícil como PC, si PM se puede resolver en tiempo polinomial, entonces PC también puede resolverse en tiempo polinomial. Estas son nuestras hipotésis por el momento

Para el caso de el problema de los maestros tenemos:

- el vector L
- el valor k con la cantidad de subgrupos a tener
- se debe organizar k cantidad de conjuntos con los maestros de agua existentes donde cada uno pertenezca a un grupo y la suma de la *ecuación 1* de menor o igual a B

Y para el caso de el problema de las cajas tenemos:

- un vector E con los espesores
- la capacidad máxima de las cajas: P
- organizar los distintos libros en k cajas, donde se busca minimizar el valor de k

Para poder demostrar esta reducción, buscamos resolver el problema de las cajas en una cantidad de pasos y llamadas polinomiales a la caja negra que resuelve el problema de los maestros.

Volviendo al problema de los maestros, la caja negra que resuelve este problema necesita como parámetros iniciales los valores: k, L y B. Entonces lo que debemos hacer para poder reducir el problema de las cajas a este es buscar que representa cada uno de estos valores en el otro. Por lo tanto, si L es el vector de maestros de agua en el problema inicial, en el problema de las cajas este es el vector con los espesores de los libros, E. Luego, k siendo la cantidad de subgrupos, esta se calcula como:

$k = sum(L)\,\%P$, donde sum() es la suma de todos los valores del vector

De esta forma, encontramos la mínima cantidad de cajas a usar y utilizamos la división euclídea que devuelve un número entero.

Finalmente, tenemos el parámetro B que es el valor de la suma que debe dar igual o menor a la **ecuación 1**. Por lo tanto, para el valor B declaramos,

$B = k * P^2$

De esta forma, P representaría el peso máximo de cada subgrupo y k representa la suma de 1 hasta n cantidad de subgrupos, que P al ser un valor constante, esto es una multiplicación.

En conclusión, tenemos las siguientes relaciones entre los parámetros que recibe la caja negra y los valores iniciales del problema de las cajas:

- $L = E$
- $k = sum(E)\,\%P$
- $B = k * P^2$

Para poder visualizar esto planteamos el siguiente ejemplo:

Si calculamos el óptimo para el problema de las cajas es: $caja\ 1 : [4, 4, 1]\ y\ caja\ 2 : [4, 3, 2]$

De esta forma, si la “caja negra"que resuelve el problema de los maestros determina que se puede formar una partición de k subgrupos tal que la suma de los cuadrados de las sumas de las

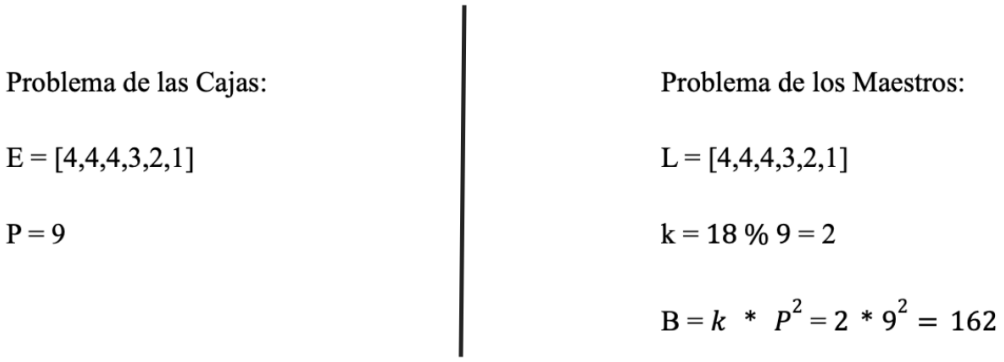

Figura 1: Ejemplo

habilidades es menor o igual a B, entonces podemos resolver el problema de las cajas utilizando la misma lógica. Para este ejemplo en específico, buscamos que la caja negra devuelva si existen dos conjuntos que dada la suma $\sum_{i=1}^{k}(\sum_{x_j \in S_i} x_j)^2$ sea menor o igual 162. Sintetizando, los parámetros que se le pasa a la caja negra son los que se muestran en la figura.

Por lo tanto, si se devuelve *true*, significa que si se pudo resolver de forma correcta el problema de las cajas. En nuestro ejemplo, la caja negra devolverá true ya que si existen dos subgrupos que sus cuadrados sumen igual o menor a B:

$$S_1 = [4, 4, 1], S_2 = [4, 3, 2] \tag{3}$$

$$\sum_{i=1}^{2}\left(\sum_{x_j \in S_i} x_j\right)^2 \leq 162 \Rightarrow \quad S_1^2 + S_2^2 \leq 162 \tag{4}$$
$$(4+4+1)^2 + (4+3+2)^2 = 162 = B$$

Se cumplen las condiciones, la caja negra devuelve true, se puede resolver el problema de las cajas.

Ahora, verificamos que no existan falsos positivos o negativos, utilizamos las siglas para más claridad donde con PC nos referinos al problema de las cajas y PM el problema de los maestros del agua:

1. Suponemos que tenemos una solución al problema de las cajas, es decir, se pueden guardar todos los libros en un número k de cajas, donde cada caja tiene una capacidad máxima P y la suma de los espesores no supera P. Si transformamos esto al PM:
   - El números de cajas k se convierte en el número de subgrupos k
   - El vector de los espesores de los libros (E) se convierte en el vector de fortalezas de los maestros del agua(L), los espesores se relacionan con las habilidades de los maestros
   - Por último, el peso P máximo de cada caja se relaciona con la suma parcial de las habilidades de cada subgrupo

   En la solución de los problemas de las cajas, si se crean k subgrupos $S_1, S_2, ..., S_k$ de la misma manera que el problema de las cajas, la suma de las habilidades de los maestros de cada subgrupo no excede al valor P y la suma de los cuadrados será menor o igual a B. Se demuestra que si hay solución al PC, hay solución al PM
2. Si suponemos no se pueden guardar todos los libros en k cajas porque se excede la capacidad, entonces no se pueden crear k subgrupos de los maestros sin excederse que la suma de habilidades sea mayor a B. Por lo tanto, si no existe solución para el PC, no existe solución para el PM.

3. Suponemos que tenemos una solución al PM, entonces si se pueden organizar los distintos maestros en diferentes en k subgrupos $S_1, S_2, ..., S_k$ donde la suma de los cuadrados de la suma de las habilidades de cada subgrupo sea menor o igual a B. Si transformamos esto en el PC tenemos:

   - Cada subgrupo $S_1, S_2, ..., S_k$ corresponde a cada una de las cajas, respectivamente.
   - Si se dividió a los maestros en k subgrupos, se utilizan k cajas para separar los libros.
   - La habilidad de cada uno de los maestros se relaciona con el espesor de cada libro.

   Si en PM se organizan los subgrupos para que la suma de los cuadrados de los subgrupos no exceda B. Si se organizan los libros de la misma manera que los subgrupos, no se excederá la capacidad P de cada caja. Se demuestra que si hay solución al PM, hay solución al PC.

4. En último lugar, si no existe solución del PM porque no se pueden organizar los maestros en k subgrupos porque la suma excede B, entonces no se pueden guardar los libros en k cajas sin excederse de la P.

Se demuestra que con nuestra reducción no existen falsos positivos ni negativos de la caja negra que resuelve el PM. Además, transformar el problema de las cajas en el problema de los maestros del agua utiliza una cantidad polinomial de pasos, ya que se recorre el vector, y se hacen las respectivas asignaciones para cada parámetro y, además, se hace un llamado a la caja de negra que resuelve el problema. Por lo tanto, la reducción es polinomial. **Se demuestra que el problema de los maestros del agua es NP-Completo.**

4. Backtracking

Implementamos un algoritmo que resuelva el problema y encuentre el óptimo utilizando *Backtracking*. Cabe mencionar que a partir de ahora buscamos minimizar la suma de cuadrados de la suma de las habilidades de los maestros de agua, quedándonos la siguiente ecuación,

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

El siguiente código es nuestro algoritmo por Backtracking,

```
def maestro_en_cada_conjunto(L,k, solucion):
    i = 0
    for maestro in L:
      solucion[i].append(maestro)
      i +=1
    coeficiente = calcular_suma(solucion)
    return coeficiente



def algoritmo_greedy(L, k):
    grupos = [[] for _ in range(k)]
    L.sort(reverse=True, key=lambda x: x[1])

    for maestro in L:
        grupos_ordenados = sorted(
            grupos, key=lambda grupo: sum((x[1] ** 2) for x in grupo))
        grupos_ordenados[0].append(maestro)

    return calcular_suma(grupos_ordenados), grupos_ordenados


def problema_del_agua(L, k):
    if k > len(L):
```

```
        return None

    sol_parcial = [[] for _ in range(k)]

    if len(L) == k:
      coeficiente = maestro_en_cada_conjunto(L,k, sol_parcial)
      return sol_parcial, coeficiente

    suma_min_mejor_gredy, mejor_solucion_greedy = algoritmo_greedy(L, k)

    mejor_solucion, suma_min_mejor = bk_problema_del_agua(
        L, k, suma_min_mejor_gredy, sol_parcial, 0)

    if mejor_solucion is None:
        return mejor_solucion_greedy, suma_min_mejor_gredy
    return mejor_solucion, suma_min_mejor



def bk_problema_del_agua(L, k, suma_min_mejor, sol_parcial, index):
    suma_min_actual = calcular_suma(sol_parcial)

    if index >= len(L):
        if len(sol_parcial) == k:
            if suma_min_actual < suma_min_mejor:
                mejor_solucion = [grupo[:] for grupo in sol_parcial]
                return mejor_solucion, suma_min_actual
        return None, suma_min_mejor

    if suma_min_actual >= suma_min_mejor:
        return None, suma_min_mejor

    mejor_solucion = None

    for i in range(k):
        sol_parcial[i].append(L[index])
        solucion, suma_min = bk_problema_del_agua(L, k, suma_min_mejor,
                                                  sol_parcial, index+1)
        if suma_min < suma_min_mejor and solucion is not None:
            suma_min_mejor = suma_min
            mejor_solucion = solucion

        sol_parcial[i].pop()

    return mejor_solucion, suma_min_mejor


def calcular_suma(solucion_parcial):
    suma_total = 0
    for grupo in solucion_parcial:
        suma_parcial = 0
        for maestro in grupo:
            suma_parcial += maestro[1]
        suma_total += suma_parcial ** 2
    return suma_total
```

En esta implementación lo que se busca es poder aplicar la mayor cantidad de podas y de esta forma obtener un algoritmo, que no solo que obtenga el óptimo, sino que sea lo más eficiente posible. Por eso decidimos implementar un algoritmo greedy que obtenga una aproximación a la solución óptima, ya que este greedy no obtiene el óptimo en el algunos casos. Este algortimo greedy es utilizado en el punto 6 (ejercicio 5 del enunciado), por lo tanto ahí discutiremos más sobre su optimalidad, implementación, y demás. Igualmente, antes de eso decidimos aplicar dos condiciones, la primera es que si se espera más subgrupos que la cantidad de maestros totales a organizar, esto no es posible. Y por otra parte, si la cantidad total de maestros es igual a la cantidad de subgrupos, se asigna cada soldado a un subgrupo y se resuelve el problema.

Luego, en primer lugar, nuestra función recibe k que es la cantidad de subgrupos en la que se debe dividir a los maestros. Además tenemos L que son los maestros de agua donde como

queríamos cumplir con los archivos de ejemplo proporcionados con la cátedra, cada maestro es una dupla donde el primer elemento es el nombre y el segundo es el atributo de fortaleza/habilidad. Por ejemplo, un posible vector puede ser,

$$L = [['Siku', 169], ['Hasook', 27], ['Desna', 76], ['Desna\ I', 156]]$$

Continuando con el algoritmo, decidimos implementar las condiciones de corte al principio donde estas son:

- el índice que recorre el vector ya es mayor que la cantidad de maestros, entonces se llego al final
- la cantidad de subgrupos en la solución parcial es igual a la pedida, k
- la suma actual ya supero a la mejor suma y todavía no se termino de iterar L

Ahora, en caso que la suma de cuadrados de la suma de los subgrupos de la solución parcial es menor que la suma de la mejor solución hasta el momento, esta se reemplaza y se asigna como la mejor solución hasta el momento. En caso que no se cumpla esta condición, significa que la solución parcial obtenida hasta el momento no es óptima y se debe buscar otra, se retorna *None* como mejor solución.

Finalmente, se itera una variable i hasta el valor de k, donde se aplica la técnica de backtracking: se guarda el soldado respectivo al indice la solución parcial para el subgrupo i, se llama recursivamente a la función aumentando el índice hasta que esta devuelve una solución mejor que la actual o *None* en caso de que no se encuentre. En caso de que sí se asigna esta como mejor solución a devolver y mejor suma hasta el momento. Luego se quita el último soldado en el subgrupo i y se vuelve a iterar. Se repite para los distintos k valores. Por último verificamos que la mejor solución que devuelve la función de backtracking no sea None, ya que significa que no se encontró ningún y que ya el algoritmo aproximación greedy dio la óptima.

Comparando con los distintos resultados de la cátedra pudimos corroborar que nuestro algoritmo obtiene siempre el óptimo.

Respecto a esto decidimos sumar las siguientes mediciones de tiempo utilizando ejemplos proporcionados por la cátedra. Decidimos comparar los tiempos de ambos algoritmos: backtracking y backtracking aproximando con greedy inicialmente. Las siguientes mediciones están en segundos.

Ejemplo	**Backtracking**	**Backtracking aproximando con Greedy**
n = 5, k = 2	5.412101745605469e-05	5.91278076171875e-05
n = 10, k = 3	0.0304720401763916	0.020236968994140625
n = 10, k = 5	2.530734062194824	0.5249168872833252
n = 11, k = 5	24.531458139419556	0.20360422134399414
n = 14, k = 3	16.259959936141968	4.364710807800293
n = 14, k = 5	6358.122680187225	195.9760389328003

Cuadro 1: Comparación de tiempos entre Backtracking y Backtracking aproximando con Greedy

Si bien todos los ejemplos los ejecutamos en nuestra terminal local, para demostrar como fue la ejecución decidimos incluir la del ejemplo n = 14 y k = 3 utilizando una notebook de colab que se puede encontrar su ejecución en el repositorio de Github.

Podemos observar que el algoritmo con la aproximación greedy es significativamente más óptimo en tiempos de ejecución, comparando el quinto ejemplo, este se ejecuto 32 veces más rápido. Se decidió no probar los tiempos de los ejemplos más chicos ya que no resultaba relevante para la medición y comparación de tiempos.

Para una mejor visualización de esta marcada diferencia, graficamos los resultados de la tabla.

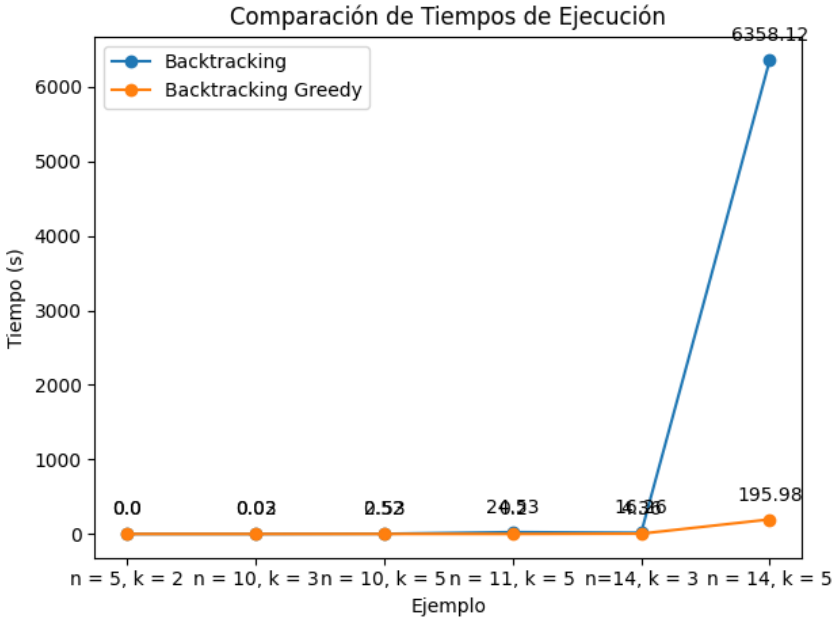

Figura 2: Comparación de tiempos de ejecución de distintos algoritmos 2

Como se puede observar en el ejemplo donde n = 14 y k = 5 la diferencia es abismal, así que decidimos graficar los otros valores y exceptuar este. De esta forma quedaría:

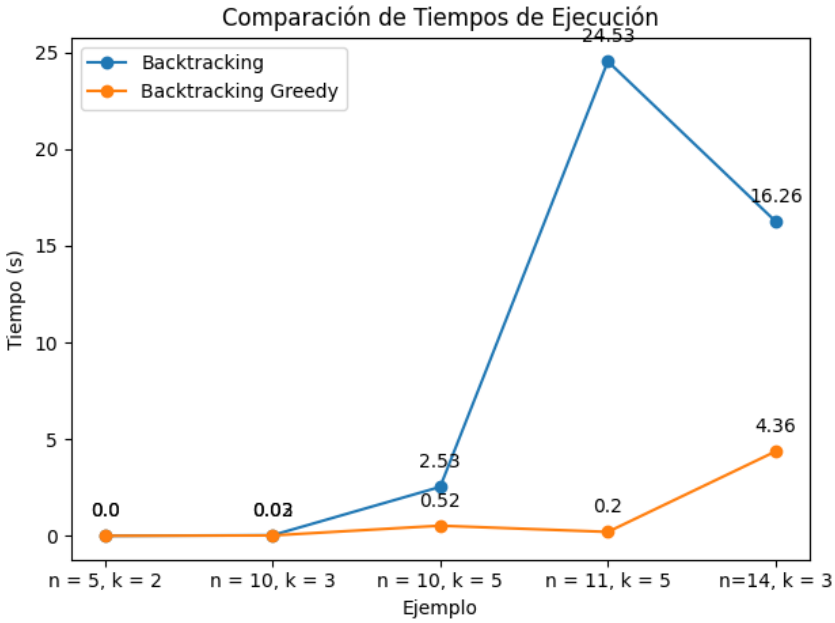

Figura 3: Comparación de tiempos de ejecución de distintos algoritmos 2

La ejecución e implementación de estos gráficos se pueden visualizar en la notebook subida al repositorio.

5. Programación Lineal

5.1. Planteo del problema

En esta sección, pasamos a analizar el problema usando la técnica matemática de Programación Lineal. La Programación Lineal se basa en especificar un objetivo como una función lineal de ciertas variables y especificar restricciones sobre los datos como igualdades o desigualdades de esas variables. Pero para este problema a resolver, claramente observemos que la función a la que queremos minimizar no es lineal, por lo que vamos a plantear una aproximación al resultado óptimo.

Vamos a tener como objetivo, minimizar la diferencia del grupo de mayor suma con el de menor suma. Es decir, si el grupo Z es el del mayor suma de habilidades de los maestros ($\sum_i Z_i$) y Y es el de menor suma, entonces se plantea como función objetivo

$$\text{mín}(\sum_i Z_i - \sum_j Y_j).$$

Introducimos las variables con las que vamos a trabajar, estas son:

- M_i: Variable continua que representa la habilidad del maestro i.
- X_{ij}: Variable binaria de decisión que indica si el maestro i está en el grupo j.
- S_j: Variable continua que representa la suma de habilidades del grupo j.
- máx S: Variable continua que representa la suma del grupo de mayor suma.
- mín S: Variable continua que representa la suma del grupo de menor suma.

Vamos a tener una variable de decisión X_{ij} para cada maestro i que pertenece a j para modelar si incluir o no este maestro al grupo. $X_{ij} = 0$ va a indicar que el maestro i esta en el grupo j y al contrario $X_{ij} = 1$ va a indicar que sí está en el grupo j. Es importante determinar que cada maestro puede estar solamente en un grupo, de esta forma:

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

Con estas variables podemos armar la variable S_j donde estaríamos definiendo las sumas de habilidades de cada grupo j. Esta suma de habilidades estaría definida como:

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

Y para obtener las variables que precisamos para la función objetivo, planteamos máx S y mín S que estas son las sumas del grupo de mayor suma y del de menor suma respectivamente.

$$\text{máx}\, S \geq S_j \quad \forall j \in \{1, \ldots, k\}$$

$$\text{mín}\, S \leq S_j \quad \forall j \in \{1, \ldots, k\}$$

Con esto llegamos a la función objetivo, minimizar la diferencia entre la máxima y la mínima suma de habilidades de los grupos:

$$\text{Min } (\text{máx}\, S - \text{mín}\, S)$$

5.2. Programación Lineal en Código

Para poder resolver este problema seguiremos utilizando el lenguaje de programación Python, precisamente mediante la implementación de la biblioteca PuLP. Esta biblioteca es la mas utilizada al momento de resolver problemas mediante Programación Lineal gracias a su facil implementacion y su facil definicion de funciones, variables y restricciones.

```
1
2  # Importamos la bibliteca PuLP
3
4  import pulp
5
6  def pl_problema_maestros_del_agua(file_path):
7      k, maestros = leer_archivo(file_path)
8      n = len(maestros)
9
10     # Inicializamos la funcion objetivo, definiendo que la queremos minimizar
11     funcion_objetivo = pulp.LpProblem("Minimizar_Diferencia_Max_Min", pulp.
       LpMinimize)
12
13     # Planteamos las variables de decision del problema
14     habilidades = [habilidad for _, habilidad in maestros]
15     X = pulp.LpVariable.dicts("X", (range(n), range(k)), cat='Binary')
16     suma_grupos = [pulp.LpVariable(f"suma_grupo_{j}", lowBound=0, cat='Continuous')
        for j in range(k)]
17     max_sum = pulp.LpVariable("max_sum", lowBound=0, cat='Continuous')
18     min_sum = pulp.LpVariable("min_sum", lowBound=0, cat='Continuous')
19
20 # Funci n objetivo: Minimizar la diferencia entre la suma del grupo de mayor suma
       con el grupo de menor suma
21     funcion_objetivo += max_sum - min_sum
22
23 # Cada maestro tiene que estar solamente en un grupo
24     for i in range(n):
25         funcion_objetivo += pulp.lpSum([X[i][j] for j in range(k)]) == 1
26
27     # Definimos las sumas de habilidades para cada grupo
28     for j in range(k):
29         funcion_objetivo += suma_grupos[j] == pulp.lpSum([habilidades[i] * X[i][j]
       for i in range(n)])
30
31     # Definimos max_sum y min_sum
32     for j in range(k):
33         funcion_objetivo += max_sum >= suma_grupos[j]
34         funcion_objetivo += min_sum <= suma_grupos[j]
35
36     # Resolver el problema
37     funcion_objetivo.solve(pulp.PULP_CBC_CMD())
```

Además, gracias a la utilización de esta librería, al momento de ejecutar el código, nos muestra por consola el resultado final del problema. Si se encontró un resultado óptimo con el valor final, el total de iteraciones por las que pasa el problema, el tiempo total del CPU (tiempo que el CPU estuvo trabajando) en segundos y el tiempo de reloj en lo que tarda en correr todo el programa.

Por ejemplo, para el caso donde trabajamos el archivo con 5 maestros del agua, y definimos un valor k = 2, la salida por consola es:

Result - Optimal solution found

Objective value: 42.00000000

Enumerated nodes: 2

Total iterations: 403

Time (CPU seconds): 0.07

Time (Wallclock seconds): 0.07

6. Algoritmo de Aproximación

```python
def aproximacion(maestros, k):
    maestros.sort(reverse= True, key=lambda x: x[1])
    grupos = [[] for _ in range(k)]

    for maestro in maestros:
        grupos_ordenados = sorted(grupos, key= lambda grupo: sum((x[1]**2) for x in grupo))
        grupos_ordenados[0].append(maestro)

    return grupos_ordenados
```

6.1. Complejidad

Para analizar la complejidad vamos a considerar a n como la cantidad de maestros y a k como la cantidad de grupos.

El algoritmo comienza ordenando los n maestros, lo cual implica una complejidad de $O(n \log n)$, seguido a esto se crean k listas vacías, $O(k)$. Luego, entra en un bucle iterando sobre todos los maestros, $O(n)$, y allí, ordena los k grupos, $O(k \log k)$, y por cada grupo ordenado se calcula la suma de los cuadrados de las habilidades, esta iteración recorre todos los elementos del grupo g, $O(g)$. Recordamos que en el peor de los casos, cada grupo puede llegar a tener hasta n elementos, siendo su complejidad $O(n)$. Por lo que, ordenar los grupos tiene una complejidad combinada $O(k\ g)$. Por último, añadir el maestro al primer grupo ordenado tiene una complejidad $O(1)$. Por lo tanto, podemos decir que cada iteración del bucle es $O(k * g)$ y como se ejecuta n veces, su complejidad queda $O(n(g * k)$, siendo así la complejidad del bucle $O(n * g * k)$, o si consideramos el peor caso del grupo donde están todos los maestros, $O(n^2 * k)$

Con las complejidades calculadas, la complejidad total seria

$O(nlogn) + O(k) + O(n * g * k) = O(n * g * k)$

Podemos concluir que la complejidad del algoritmo de aproximación es $O(n * g * k)$.

6.2. Análisis de la aproximación

Tras haber calculado las mediciones de tiempo del algoritmo por backtracking y mediciones con el algoritmo de aproximación, se puede ver una gran diferencia entre ambas.

Se puede ver para el caso en que $n = 10$ y $k = 5$ obtuvimos que: Suma Exacta: 355882 Tiempo Exacto: 7.498732328414917

Suma Aproximada: 366578 Tiempo Aproximado: 6.985664367675781e-05

Se puede observar que la solución exacta tardo más en ejecutarse pero dio el resultado correcto.

A su vez calculamos la relación de aproximación $r(A)$, para determinarla, realizamos dichas medidas. La relación $r(A)$ se define como la proporción entre la solución aproximada $A(I)$ y la solución óptima $z(I)$.

Esta relación proporciona una medida de cuanto puede desviarse la solución aproximada de la solución óptima en el peor de los casos, lo cual es útil para evaluar la efectividad del algoritmo de aproximación en términos de su capacidad para encontrar soluciones cercanas a la óptima.

Para volúmenes de datos grandes, el algoritmo de aproximación logra dar resultados efectivos, en poco tiempo.

En esta notebook que se encuentra en el repositorio se pueden observar todos los ejemplos ejecutados, su soluciones, ya sea aproximada como exacta y sus tiempos de ejecución.

6.3. Conclusión

En definitiva, durante este trabajo practico, estuvimos trabajando con el Problema de la Tribu del Agua. Un problema que al plantearlo como problema de decisión pertenece a la clase de complejidad NP-completo. En un principio pudimos demostrar mediante un certificador que este problema es de clase NP, consecuentemente después de esa verificación pudimos también demostrar su NP-completitud, por medio de una reducción polinomial del problema de los libros y las cajas a este.

Luego de estas demostraciones, planteamos la resolución del problema de optimización mediante la técnica de Backtracking, la implementación de un problema de Backtracking utilizando al mismo tiempo una estrategia Greedy. Con estos dos últimos denotamos la optimalidad mayor a la hora de ejecutarlo con la resolución teniendo la aproximación Greedy. Pudimos también resolver, por un tema de alta complejidad del problema, una aproximación utilizando la técnica matemática de Programación Lineal, planteando su función objetivo, sus variables y sus restricciones. Por ultimo, realizamos el algoritmo de aproximación propuesto en el enunciado, con su análisis de ejecución y de complejidad correspondiente.

Por lo tanto, damos como finalizado el análisis de este problema. Denotamos que si bien resolver problemas de clase NP-completos es una gran dificultad en la teoría de algoritmos, de todas maneras se puede encontrar una solución óptima mediante la utilización de las técnicas vistas anteriormente. Además, destacamos la importancia de realizar una aproximación a la hora de resolver problemas de esta clase para poder resolverlo en un tiempo razonable, especialmente los de gran tamaño, esto lleva a ser mas eficiente en cuanto a recursos y pueda ser mas aplicable al momento de tener un caso real.

7. Anexo

7.1. Demostración que el problema de los maestros del agua es NP-Completo

En esta sección demostraremos que el problema de los maestros del agua es NP-Completo (NP-C), para poder realizar esto debemos buscar un problema NP-C más "sencillo"que este, lo denominamos *problema X*. Por lo tanto, debemos reducir este problema X al problema de los maestros y poder resolver X usando la "caja negra"que resuelve el problema de los maestros. Cuando mencionamos la "caja negra"nos referimos a un problema de decisión donde la "caja"devuelve los valores booleanos true o false. Entonces al mencionar esta caja lo que buscamos es que el problema de decisión sea si existe una partición de k subgrupos, tal que la suma total calculada en la *ecuación 1* sea igual a B.

Por lo tanto, decidimos reducir el *k-partition problem* a el problema de los maestros del agua (PMA), pero para simplificar no hace falta considerar un k genérico, usamos $K = 2$.

El enunciado de 2-Partition es: Dado un conjunto de valores $T = \{t_1, t_2, t_3, ..., t_n\}$ se busca dividirlo en dos subconjuntos donde la suma de sus elementos de cada subconjunto sea igual para ambos.

7.2. Reducción 2-Partition Problem a PMA

Dado lo visto en la clase: *2-Partition* es un problema *NP-Completo*, y, por lo tanto, buscamos reducir este a PMA. Para esto utilizamos la caja negra de PMA que devolverá true o false si se puede resolver el problema. Por lo tanto, buscamos realizar la siguiente reducción:

$$2 - Partition \leq_p Problema\ de\ los\ Maestros\ del\ Agua \tag{6}$$

En el caso de *2-Partition*, este recibe $T = \{t_1, t_2, t_3, ..., t_n\}$ donde cada t_i representa un número. De esta forma reducimos que cada t_i va a representar un maestro del agua, por lo tanto tenemos que $T = L$ cada t_i corresponde a un l_i. Continuando, el PMA recibe un valor k de cuántos subconjuntos de maestros va a haber, en este caso k va a valer siempre 2 porque se busca siempre dividir en dos subconjuntos. Finalmente, el último parámetro que recibe la caja negra de PMA es el valor B de cuanto va a dar la suma total, para poder realizar la reducción consideramos como la suma de ambos subconjunto debería ser $\frac{\sum_{i=1}^{n} t_i}{2}$, para la reducción consideramos que el cuadrado de la suma de cada subconjunto es $(\frac{\sum_{i=1}^{n} t_i}{2})^2$ y lo debemos multiplicar por k (k=2) así representamos la suma de los subconjuntos, que en este caso son siempre dos, es decir, $2 * (\frac{\sum_{i=1}^{n} l_i}{2})^2$.

En conclusión, estos son los parametros que recibirá la caja negra que resuelve el problema PMA para poder resolver la reducción de *2-partition*:

- $T = L$, $\{t_1, t_2, t_3, ..., t_n\} = \{l_1, l_2, l_3, ..., l_n\}$ donde cada $t_i = l_i$ ya que i vale de 0 hasta n siendo n la cantidad de elementos en el conjunto.
- $k = 2$, la cantidad k de subgrupos a dividir es siempre dos
- $B = 2 * (\frac{\sum_{i=1}^{n} l_i}{2})^2$, donde B es el valor total que debe dar la suma total

Para poder visualizar esto planteamos el siguiente ejemplo:

Tenemos el conjunto de valores $T = \{2, 3, 7, 6, 2, 1, 1\}$ y queremos separar el conjunto a subconjuntos utilizando 2-Partition. Por lo tanto planteamos para la reducción de 2-Partition a PMA,

1. Si $T = L$, $L = \{2, 3, 7, 6, 2, 1, 1\}$, $k = 2$ y $B = 2 * (\frac{\sum_{i=1}^{n} l_i}{2})^2 = 2 * (\frac{22}{2})^2 = 242$
2. Al utilizar la caja negra que resuelve el PMA para este problema, esta va a devolver *True*, ya que si se puede dividir el conjunto en dos y que la suma de B

En este caso el conjunto se podría dividir como: $T_1 = \{2, 7, 2\}$ y el otro como $\{3, 6, 1, 1\}$, por lo tanto la reducción fue correcta.

Es decir se puede resolver un problema de 2-Partition utilizando la caja negra que resuelve el PMA.

Ahora verificamos que no existan falsos positivos ni negativos para la reducción:

1. Suponemos que tenemos una solución del problema 2-Partition, es decir, se puede dividir a x conjunto en 2 subconjuntos donde la suma de todos los elementos de cada uno es igual. Ahora si transformamos eso al PMA tenemos que:

   - El conjunto inicial T de 2-Partition es el conjunto L en PMA.
   - El número k de subconjuntos a formar con los elementos del conjunto L es 2
   - Por último, la suma total B a dar es igual a $2 * (\frac{\sum_{i=1}^{n} l_i}{2})^2$.

   En el problema de 2-Partition se va a crear dos subconjutos los cuáles van a ser los mismos para PMA ya que también crea dos subconjuntos. Y la suma de ambos subconjuntos en PMA van a ser iguales entre sí ya que el valor de la suma, B, se define para que de así.

   Se demuestra si hay solución en *2-Partition*, hay solución en PMA.

2. Sin embargo, si existe una solución para PMA donde dado un conjunto L, se busca dividirlo en $k = 2$ subconjutos y que la suma de los cuadrados de cada subconjunto l_i de $B = 2*(\frac{\sum_{i=1}^{n} l_i}{2})^2$, transformando esto para 2-Partition tenemos que:

   - El conjunto $L = T$
   - Como se quiere dividir L en $k = 2$ subconjuntos, esto cumple con el problema de $2 - Partition$ que busca dividir en 2 subconjuntos al conjunto T.

   Finalmente, si existe una solución de 2 subconjuntos l_1 y l_2 para el PMA, entonces estos dos subconjuntos correponden a t_1 y t_2 que también son solución para 2-Partition. Por lo tanto, se pudo demostrar que si hay solución para PMA, hay solución para 2-Partition.

En conclusión, se pudo demostrar que no existen falsos positivos ni negativos en la reducción de 2-Partition a PMA. También, transformar el problema 2-Partition a PMA utiliza una cantidad polinomial de pasos, ya que se asignan elementos (vectores) y se calcula una suma, y se llama una vez a la caja negra, se concluye que la reducción es polinomial. Finalmente, se demuestra que el problema *2-Partition* es al menos tan difícil que PMA, es decir, **el problema de los maestros del agua es NP-Completo**.

7.3. Programacion Lineal

7.3.1. Planteo del problema

Variables con las que vamos a trabajar, estas son:

- M_i: Constante que representa la habilidad del maestro i.
- X_{ij}: Variable binaria de decisión que indica si el maestro i está en el grupo j.
- S_j: Variable continua que representa la suma de habilidades del grupo j.
- máx S: Variable continua que representa la suma del grupo de mayor suma.
- mín S: Variable continua que representa la suma del grupo de menor suma.

7.3.2. Programación Lineal en Código

Para poder corroborar la correctitud del código, planteamos un método de reconstrucción.

```
    # Imprimir el estado del problema
    print(f"El estado de la funcon resulta: {pulp.LpStatus[funcion_objetivo.status
    ]}")
    print(f"Los grupos quedan conformados como:")

    # Imprimir los grupos
    if pulp.LpStatus[funcion_objetivo.status] == 'Optimal':
        grupos = [[] for _ in range(k)]
        for i in range(n):
            for j in range(k):
                if pulp.value(X[i][j]) == 1:
                    grupos[j].append(maestros[i][0])

        for j in range(k):
            print(f"Grupo {j+1}: {', '.join(grupos[j])}")
```

Gracias a este algoritmo, podemos verificar que es correcta la resolución del problema mediante esta aproximación. Sin embargo, puede que no sea la mejor solución en cuanto a optimalidad y tiempos de ejecución, por eso, realizaremos una comparación de tiempo entre esta estrategia y el método de Backtracking.

7.3.3. Mediciones de tiempo

Planteamos un código para poder recorrer los archivos que queremos comparar, y que resuelva cada set de datos con la técnica de Backtracking y con la de Programación Lineal y realice la medición de tiempo de ejecución para cada archivo.

```
def procesar_archivos(folder_path):
    datos = []
    for archivo in os.listdir(folder_path):
        if archivo.endswith(".txt"):
            file_path = os.path.join(folder_path, archivo)
            k, maestros = leer_archivo(file_path)

            inicio = time.time()
            problema_del_agua(maestros, k)
            final = time.time()
            tiempo_bt = final - inicio

            inicio2 = time.time()
            pl_problema_maestros_del_agua(k, maestros)
            final2 = time.time()
            tiempo_pl = final2 - inicio2

            datos.append({
                "Archivo": archivo,
                "Tiempo Backtracking": tiempo_bt,
                "Tiempo PL": tiempo_pl
            })

    return datos
```

Con este algoritmo, almacenamos los tiempos de ejecución de cada técnica en un archivo CSV. Pudimos trabajar y llegar a estos resultados con estos archivos:

Archivo	Tiempo Backtracking	Tiempo Programación Lineal
5_2.txt	0.0	0.13415074348449707
10_3.txt	0.029208898544311523	0.3789949417114258
10_5.txt	2.409165859222412	2.938222646713257
11_5.txt	26.319823265075684	1.2530674934387207
14_3.txt	6.073249340057373	3.207059621810913
14_4.txt	317.07192039489746	11.775965213775635

Cuadro 2: Comparación de tiempos entre Backtracking y Programación Lineal

Para una mejor visualización de este cuadro armamos un gráfico para comparar los tiempos de ejecución.

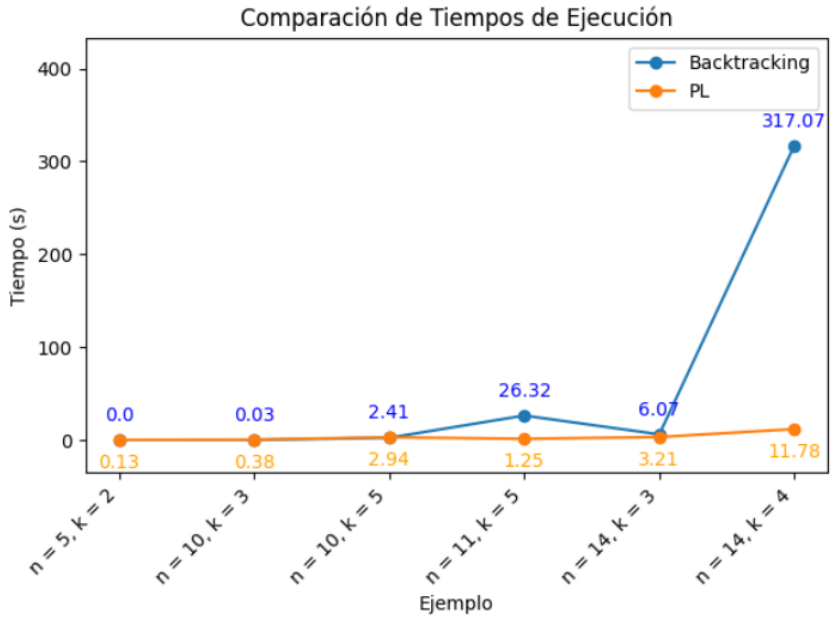

Figura 4: Comparación de tiempos de ejecución de distintos métodos

Podemos determinar claramente, a diferencia de la suposición que se tenia anteriormente, como el algoritmo por Programación Lineal resuelve muchísimo mas rápido el problema para los casos donde la cantidad de grupos a buscar k es mayor.

7.4. Algoritmo de Aproximación

Para llevar a cabo el algoritmo de aproximación elegimos usar un heap debido a su eficiencia en la gestión de prioridades. Al utilizar un heap, podemos extraer y reinsertar el grupo con la menor suma en tiempo $O(log\ k)$, lo que hace que el proceso sea eficiente incluso cuando tenemos un numero de grupos k grande. Esto es esencial para garantizar que el algoritmo sea escalable y capaz de manejar grandes volúmenes de datos de manera eficiente mientras mantiene los grupos equilibrados en cada paso.

El código en Python para este algoritmo es:

```python
def calcular_suma(grupo):
    suma_parcial = sum(maestro[1] for maestro in grupo)
    return suma_parcial ** 2

def aproximacion(maestros, k):
    maestros.sort(reverse=True, key=lambda x: x[1])

    heap = [(0, []) for _ in range(k)]
    heapq.heapify(heap)

    for maestro in maestros:
        suma, grupo = heapq.heappop(heap)
        grupo.append(maestro)
        nueva_suma = calcular_suma(grupo)
        heapq.heappush(heap, (nueva_suma, grupo))

    grupos_ordenados = [grupo for suma, grupo in heap]

    return grupos_ordenados
```

7.4.1. Complejidad

Para analizar la complejidad vamos a considerar a n como la cantidad de maestros y a k como la cantidad de grupos.

El algoritmo comienza ordenando los n maestros, lo cual implica una complejidad de $O(n \log\ n)$, seguido a esto se inicializa un heap con k elementos, $O(k)$. Luego, entra en un bucle iterando sobre todos los maestros, $O(n)$, y allí hace 4 ejecuciones:

1. Extrae el grupo con la suma más baja del heap, $O(log\ k)$.
2. Agrega el maestro al grupo, $O(1)$.
3. Recalcula la suma del grupo, siendo $O(g)$ donde g es el tamaño del grupo
4. Actualiza la suma del grupo, $O(log\ k)$.

Por lo tanto, podemos decir que cada iteración del bucle es $O(log\ k+g)$. Sin embargo, en la práctica el tamaño de g crece linealmente con el número de iteraciones, pero el costo de calcular la suma se distribuye uniformemente. Siendo así que, en promedio, calcular la suma tiene un costo constante $O(1)$. Por ende, cada iteración del bucle tiene una complejidad $O(log\ k)$ y como se ejecuta n veces, su complejidad queda $O(n\ log\ k)$.

Con las complejidades calculadas, la complejidad total seria

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

7.4.2. Análisis de la aproximación

Tras haber calculado las mediciones de tiempo del algoritmo por backtracking y mediciones con el algoritmo de aproximación, se puede ver una gran diferencia entre ambas.

Para una mejor visualización, armamos un gráfico comparando los tiempos de ejecución.

Archivo	Tiempo Aproximación	A(I)	Tiempo Backtracking	z(I)	r(A)
5_2.txt	0.000026	1894340	0.000110	1894340	1.000000
10_3.txt	0.000021	385249	0.051945	385249	1.000000
10_5.txt	0.000038	355882	1.337772	355882	1.000000
11_5.txt	0.000035	2906564	0.626441	2906564	1.000000
14_3.txt	0.000037	15664276	3.761700	15659106	1.000330
14_4.txt	0.000068	15292085	32.498133	15292055	1.000002

Cuadro 3: Comparación de tiempos entre Backtracking y Aproximación

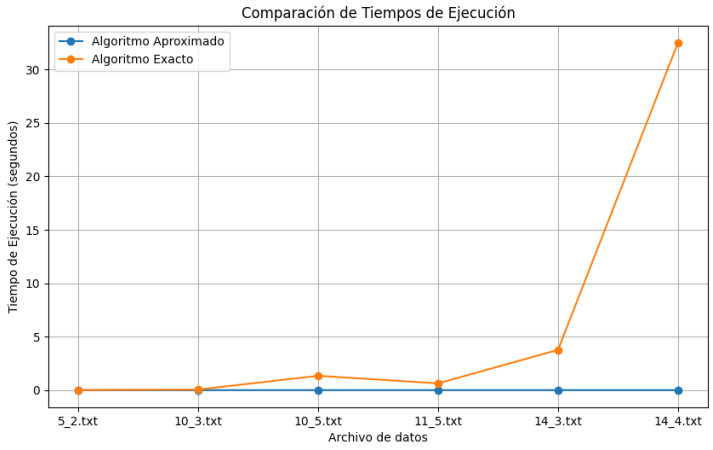

Figura 5: Comparación de tiempos de ejecución

La comparación de tiempos de ejecución entre los algoritmos de backtracking y aproximación muestra diferencias notables en términos de escalabilidad y eficiencia. El algoritmo de aproximación presenta una excelente escalabilidad, manteniendo tiempos de ejecución consistentemente bajos independientemente del tamaño del archivo. Por ejemplo, en el archivo "14_4.*txt*", el tiempo de ejecución del algoritmo de aproximación es de solo 0.000068 segundos. Mientras que, el algoritmo de backtracking muestra una escalabilidad deficiente, con tiempos de ejecución que aumentan exponencialmente a medida que el tamaño del archivo y la complejidad del problema crecen. Para el mismo archivo "14_4.*txt*", el tiempo de backtracking alcanza los 32.498133 segundos.

En términos de eficiencia, el algoritmo de aproximación es altamente eficiente. En todos los archivos analizados, el tiempo de ejecución se mantiene por debajo de 0.0001 segundos, lo cual demuestra la rapidez del algoritmo. Por otro lado, la eficiencia del algoritmo de backtracking es considerablemente menor, especialmente con archivos más grandes. Los tiempos de ejecución pueden superar los 30 segundos, como se observa en el archivo "14_4.*txt*", lo que hace que el algoritmo de backtracking sea menos práctico para problemas de gran escala.

A su vez calculamos la relación de aproximación $r(A)$, para determinarla, realizamos dichas medidas. La relación $r(A)$ se define como la proporción entre la solución aproximada $A(I)$ y la solución óptima $z(I)$.

Esta relación proporciona una medida de cuanto puede desviarse la solución aproximada de la solución óptima en el peor de los casos, lo cual es útil para evaluar la efectividad del algoritmo de aproximación en términos de su capacidad para encontrar soluciones cercanas a la óptima.

Para demostrar la eficiencia del algoritmo de aproximación, la evaluamos en volúmenes de datos grandes, realizamos mediciones con los archivos proporcionados por la cátedra. Los resultados obtenidos muestran que el algoritmo de aproximación ofrece resultados precisos en tiempos muy cortos.

Archivo	Tiempo Aproximación	A(I)	z(I)
17_10.txt	0.000040	5430512	5427764
18_6.txt	0.000030	10325588	10322822
18_8.txt	0.000029	12000279	11971097
20_4.txt	0.000031	21083935	21081875
20_5.txt	0.000029	16838539	16828799
20_8.txt	0.000030	11423826	11417428

Cuadro 4: Comparación entre tamaño de datos y tiempo

Como se observa en la tabla, el algoritmo de aproximación permite procesar grandes volúmenes de datos en tiempos extremadamente reducidos, con una diferencia mínima entre los resultados aproximados ($A(I)$) y los resultados exactos ($z(I)$). Esto demuestra la efectividad y eficiencia del algoritmo en contextos con grandes entradas de datos.

En conclusión, el análisis de tiempos muestra claramente que el algoritmo de aproximación es significativamente más rápido y escalable en comparación con el algoritmo de backtracking. Aunque el algoritmo de aproximación es mucho más eficiente, no compromete la precisión, manteniendo una relación $r(A)$ cercana a 1. Esto lo convierte en una excelente opción para problemas grandes y complejos, donde el tiempo de ejecución es crítico. Para aplicaciones prácticas en las que el tamaño del problema puede ser grande, el algoritmo de aproximación ofrece una solución viable y efectiva, proporcionando resultados casi óptimos en una fracción del tiempo necesario para obtener la solución exacta mediante backtracking.