.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchhwald - Genender

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

3 de julio de 2024

Ana Gabriela Gutson	Lucas Franciulli	Mariana Juarez Goldemberg
105853	107059	108441

Índice general

1. Introducción

1.1. Problema

Es el año 95 DG. La Nación del Fuego sigue su ataque, esta vez hacia la Tribu del Agua, luego de una humillante derrota a manos del Reino de la Tierra, gracias a nuestra ayuda. La tribu debe defenderse del ataque.

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

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

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

Para que los grupos estén parejos, lo que buscaremos es minimizar la adición de los cuadrados de las sumas de las fuerzas de los grupos. Es decir:

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

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

1.2. Consigna

Para los primeros dos puntos, considerar la versión de decisión del Problema de la Tribu del Agua:

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

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

1. Demostrar que el Problema de la Tribu del Agua se encuentra en NP.
2. Demostrar que el Problema de la Tribu del Agua es, en efecto, un problema NP-Completo. Si se hace una reducción involucrando un problema no visto en clase, agregar una (al menos resumida) demostración de que dicho problema es NP-Completo.

3. Escribir un algoritmo que, por backtracking, obtenga la solución óptima al problema en la versión de optimización planteada originalmente. Generar conjuntos de datos para corroborar su correctitud, así como tomar mediciones de tiempos.
4. Escribir un modelo de programación lineal que resuelva el problema de forma óptima. Ejecutarlo para los mismos conjuntos de datos para corroborar su correctitud. Tomar mediciones de tiempos y compararlas con las del algoritmo que implementa Backtracking.
   - Obtener la solución óptima al problema (más allá del planteo, demorará bastante en su ejecución). Considerar que esto implicará agregar bastantes variables de decisión.
   - Que el objetivo sea 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 busca minimizar $\sum_i Z_i - \sum_j Y_j$. En este caso, la solución será una aproximación al resultado óptimo.
5. El Maestro Pakku nos propone el siguiente algoritmo de aproximación: 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.

   Implementar dicho algoritmo, analizar su complejidad y analizar cuán buena aproximación es. Para esto, considerar la relación $A(I)/z(I) \leq r(A)$ para todas las instancias posibles, donde $A(I)$ es la solución aproximada, $z(I)$ es una solución óptima y $r(A)$ es una relación definida previamente.

   Realizar mediciones utilizando el algoritmo exacto y la aproximación, con el objetivo de definir dicha relación. Realizar también mediciones que contemplen volúmenes de datos ya inmanejables para el algoritmo exacto, a fin de corroborar empíricamente la cota calculada anteriormente (implementando para conjuntos de datos cuya solución se sepa de antemano).
6. Implementar alguna otra aproximación (o algoritmo greedy) que les parezca de interés. Comparar sus resultados con los dados por la aproximación del punto anterior. Indicar y justificar su complejidad.
7. Agregar cualquier conclusión que parezca relevante.

2. Análisis del Problema

El objetivo es dividir a los Maestros Agua en grupos de manera que se minimice la adición de los cuadrados de las sumas de las fuerzas de los grupos. Esto es importante porque se busca mantener un ataque constante y efectivo contra la Nación del Fuego.

2.1. Variables

- k: cantidad de grupos a armar.
- x_i: fuerza/habilidad/maestría de cada uno de los maestros agua i.
- Condición para que los grupos sean parejos (según especifique cada punto)

2.2. Ejemplo para problema original

Recordemos que la condición original de "parejamiento"de los grupos se rige por:

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

Supongamos que tenemos 3 maestros de agua con las siguientes fuerzas:

$$x_1 = 2, \quad x_2 = 3, \quad x_3 = 4$$

y tenemos que dividirlos en 2 grupos.

Las combinaciones posibles son:

1. Grupo 1: x_1, x_2 - Grupo 2: x_3
2. Grupo 1: x_1, x_3 - Grupo 2: x_2
3. Grupo 1: x_2, x_3 - Grupo 2: x_1

Para cada una de estas combinaciones, calculamos la suma de los cuadrados de las sumas de las fuerzas de los grupos.

1. $(2+3)^2 + 4^2 = 41$
2. $(2+4)^2 + 3^2 = 45$
3. $(3+4)^2 + 2^2 = 53$

Después de calcular estas sumas, seleccionamos la combinación que minimiza la adición de los cuadrados de las sumas de las fuerzas de los grupos. En este caso, la mejor división encontrada sería la primera conformada por:

Grupo 1: Maestro 1, Maestro 2
Grupo 2: Maestro 3

3. Demostración NP y NP Completo

La partición de los maestros de agua en k subgrupos cumplirá la siguiente condición:

Todos los k subgrupos armados deberán cumplir que la suma de los cuadrados de las fuerzas de los integrantes sea menor al valor B.

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

Donde cada elemento x_i debe estar asignado a un grupo y solo un grupo.

3.1. El Problema de la Tribu del Agua está en NP

Para demostrar que el Problema de la Tribu del Agua está en NP, debemos mostrar que, dado un certificado (una partición de los maestros en k grupos), podemos verificar en tiempo polinomial si esta partición cumple con la condición de que la suma de los cuadrados de las fuerzas de los grupos es menor o igual a B.

El algoritmo funciona de la siguiente manera:

1. Primero, se verifica si la longitud de la partición es igual a la longitud de las fuerzas de los maestros. Si no es así, se devuelve False inmediatamente.
2. Se inicializa una lista llamada `fuerzas_particiones` con k ceros para almacenar la suma de las fuerzas de los maestros en cada partición.
3. Se itera sobre cada maestro y se suma su fuerza a la partición correspondiente. Esta operación se realiza en tiempo lineal respecto al número de maestros.
4. Luego, se calcula la suma de los cuadrados de las fuerzas de cada partición. Esta operación se realiza en tiempo lineal respecto al número de particiones.
5. Finalmente, se compara la suma de los cuadrados con el límite B. Si la suma es menor o igual a B, se devuelve True; de lo contrario, se devuelve False.

```python
def verificador_tribu_agua(fuerzas, k, B, particion):
    if len(particion) != len(fuerzas): # O(1)
        return False
    fuerzas_particiones = [0] * k
    for i, fuerza in enumerate(fuerzas): # O(n)
        particiones_index = particion[i]
        fuerzas_particiones[particiones_index] += fuerza
    suma_cuadrados = sum(f ** 2 for f in fuerzas_particiones) # O(k)
    return suma_cuadrados <= B
```

El orden de la verificación será $O(n + k)$, lo que es polinomial en el tamaño de la entrada, por lo tanto, el Problema de la Tribu del Agua está en NP.

3.2. El Problema de la Tribu del Agua es NP-Completo

Ya hemos demostrado que el Problema de la Tribu del Agua está en NP. Ahora, demostraremos que es NP-Completo mediante una reducción desde el Problema de la Mochila (Knapsack).

El Problema de la Mochila se define como sigue: "Dado un conjunto de n objetos, cada uno con un peso w_i y un valor v_i, determinar si se puede seleccionar un subconjunto de estos objetos cuyo peso total no exceda W y cuyo valor total sea al menos V."

Vamos a reducir el Problema de la Mochila al Problema de la Tribu del Agua. La idea básica es mapear los pesos de los objetos a las fuerzas de los maestros y considerar la capacidad de la mochila como el límite B.

Para esta reducción, consideramos una instancia del Problema de la Mochila con:

- n objetos con pesos $w_1, w_2, \ldots, w_n$.
- Una capacidad de mochila W.
- Un valor objetivo V.

Construimos una instancia del Problema de la Tribu del Agua como sigue:

- Los valores de fuerza x_i son iguales a los pesos w_i de los objetos.
- El número de grupos k es 1.
- El límite B es igual a W^2.

La solución al Problema de la Mochila se traduce a nuestro problema si podemos particionar los maestros en un solo grupo de tal manera que la suma de las fuerzas al cuadrado no exceda B. Esto es equivalente a verificar si podemos llenar la mochila con un peso total no mayor que W.

Por lo tanto, si podemos resolver el Problema de la Tribu del Agua en tiempo polinomial, también podríamos resolver el Problema de la Mochila en tiempo polinomial, lo que implica que el Problema de la Tribu del Agua es NP-Completo.

3.3. Conclusión

Hemos demostrado que el Problema de la Tribu del Agua está en NP y que es NP-Completo mediante una reducción del Problema de la Mochila. Por lo tanto, hemos establecido que el Problema de la Tribu del Agua es NP-Completo.

4. Backtracking

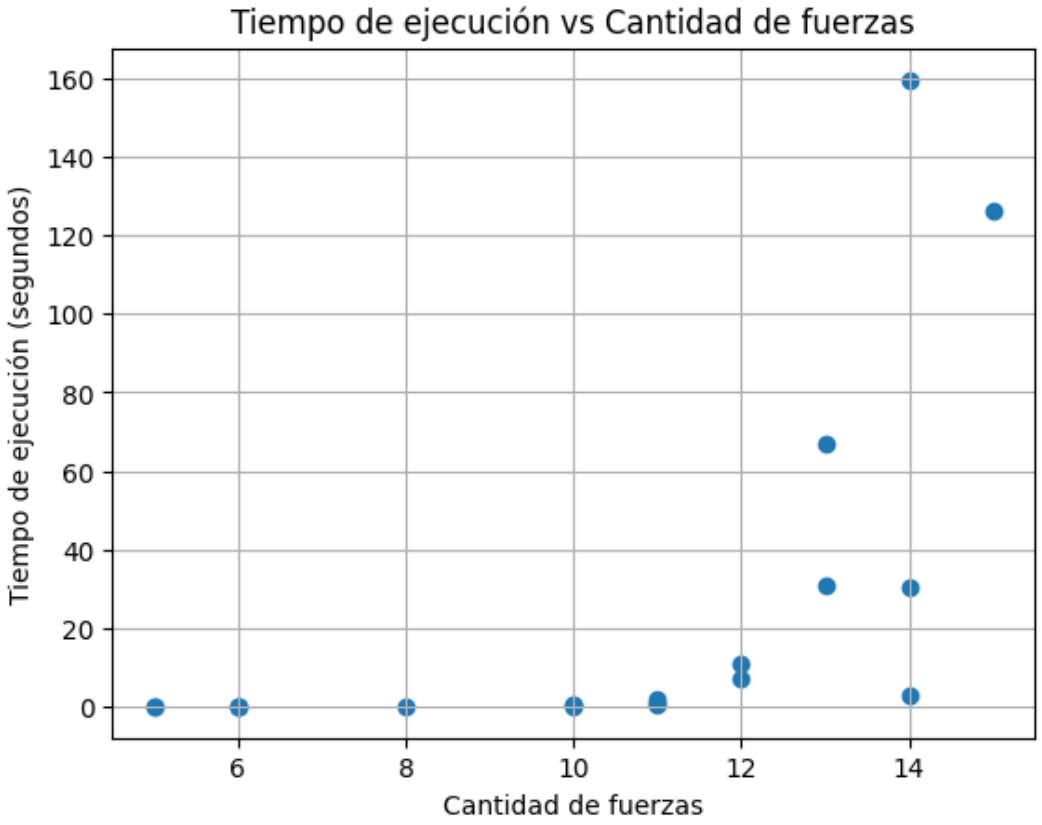
Para obtener la solución óptima al Problema de la Tribu del Agua, utilizamos un algoritmo de backtracking que explora todas las posibles particiones de los maestros en k grupos.

4.1. Definición del Algoritmo

El algoritmo de backtracking explora todas las combinaciones posibles de asignaciones de maestros a grupos para encontrar la partición que minimiza la suma de los cuadrados de las sumas de las fuerzas de los grupos.

```python
def backtracking_tribu_agua(fuerzas, k):
    n = len(fuerzas)
    mejores_grupos = None
    mejor_costo = float('inf')

    def calcular_costo(grupos):
        return sum(sum(grupo)**2 for grupo in grupos)

    def backtrack(division_actual, indice):
        nonlocal mejores_grupos, mejor_costo
        if indice == n:
            costo_actual = calcular_costo(division_actual)
            if costo_actual < mejor_costo:
                mejor_costo = costo_actual
                mejores_grupos = [list(grupo) for grupo in division_actual]
            return

        for grupo in division_actual:
            grupo.append(fuerzas[indice])
            backtrack(division_actual, indice + 1)
            grupo.pop()

        if len(division_actual) < k:
            nuevo_grupo = [fuerzas[indice]]
            division_actual.append(nuevo_grupo)
            backtrack(division_actual, indice + 1)
            division_actual.pop()

    backtrack([], 0)
    return mejores_grupos, mejor_costo
```

4.2. Mediciones



4.3. Conclusión

Hemos desarrollado un algoritmo de backtracking para obtener la solución óptima al Problema de la Tribu del Agua. Este algoritmo explora todas las posibles particiones para encontrar la mejor asignación de maestros a grupos.

5. Programación Lineal

5.1. Modelado del Problema

Debemos transformar el problema para poder resolverlo con programación lineal entera. Definiremos las siguientes variables:

- x_{ij}: Variable binaria que indica si el maestro i está asignado al grupo j.
- z_j: Variable de decisión que representa la suma de habilidades del grupo j.
- y: Variable de decisión que representa la diferencia entre la suma del grupo de mayor habilidad y la del grupo de menor habilidad.

Las restricciones del modelo son las siguientes:

Cada maestro debe ser asignado a un único grupo:

$$\sum_{j=1}^{k} x_{ij} = 1, \quad \forall i$$

La suma de las habilidades en cada grupo debe ser igual a z_j:

$$z_j = \sum_{i=1}^{n} x_i \cdot x_{ij}, \quad \forall j$$

La variable y debe ser mayor o igual que la diferencia entre el grupo de mayor suma y el de menor suma:

$y \geq z_j - z_{j'}, \quad \forall j, j'$

Las variables x_{ij} son binarias:

$x_{ij} \in \{0, 1\}, \quad \forall i, j$

La función objetivo del modelo será minimizar la variable y.

Para resolver este problema utilizando un solver de programación lineal entera, se pueden usar herramientas como Gurobi, CPLEX o el solver de programación lineal de Python (PuLP). A la hora de implementar el algoritmo, utilizamos el solver que hemos visto en clase (PuLP):

```
from pulp import LpProblem, LpMinimize, LpVariable, lpSum, PULP_CBC_CMD

def solve_tribu_agua(fuerzas, k):
    n = len(fuerzas)

    # Crear el problema
    prob = LpProblem("Problema de la Tribu del Agua", LpMinimize)

    # Variables de decision
    x = LpVariable.dicts("x", (range(n), range(k)), cat='Binary')
```

```
    z = LpVariable.dicts("z", range(k))
    y = LpVariable("y")

    # Restricciones
    for i in range(n):
        prob += lpSum(x[i][j] for j in range(k)) == 1

    for j in range(k):
        prob += z[j] == lpSum(fuerzas[i] * x[i][j] for i in range(n))

    for j in range(k):
        for j_prime in range(k):
            if j != j_prime:
                prob += y >= z[j] - z[j_prime]
                prob += y >= z[j_prime] - z[j]

    # Funcion objetivo
    prob += y

    # Resolver el problema
    prob.solve(PULP_CBC_CMD())

    # Obtener los resultados
    grupos = [[] for _ in range(k)]
    for i in range(n):
        for j in range(k):
            if x[i][j].varValue > 0:
                grupos[j].append(fuerzas[i])

    return grupos, y.varValue
```

Esta implementación crea y resuelve el modelo de programación lineal entera para el Problema de la Tribu del Agua, asignando maestros a grupos de manera que se minimice la diferencia entre el grupo de mayor habilidad y el de menor habilidad.

5.2. Comparación de Tiempos de Ejecución

Para comparar la eficiencia del algoritmo de programación lineal con el algoritmo de backtracking, ejecutamos ambos con los mismos conjuntos de datos y tomamos mediciones de los tiempos de ejecución.

5.2.1. Conjuntos de Datos

Usamos los siguientes conjuntos de datos para las pruebas:

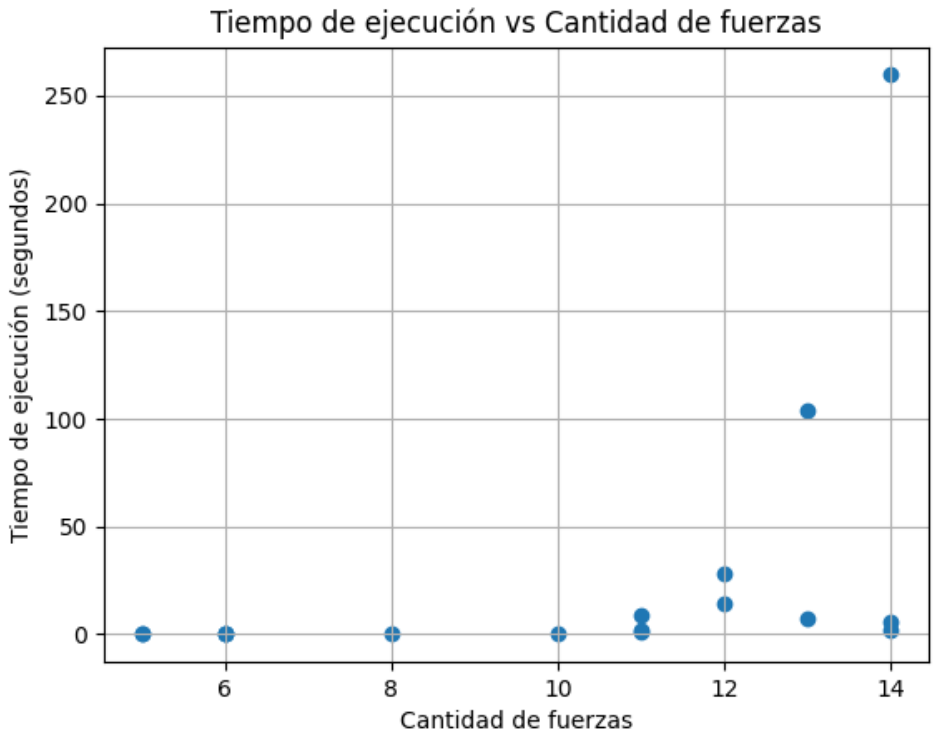
- Conjunto de datos 1: [2, 3, 4, 5, 6, 7]
- Conjunto de datos 2: [10, 15, 20, 25, 30, 35, 40, 45, 50]
- Conjunto de datos 3: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

5.2.2. Resultados y Tiempos de Ejecución

Conjunto de Datos	Tiempo (Programación Lineal)	Tiempo (Backtracking)
Conjunto de datos 1	2.7179718017578125e-05 segundos	0.00013184547424316406 segundos
Conjunto de datos 2	2.384185791015625e-05 segundos	0.0077266693115234375 segundos
Conjunto de datos 3	4.506111145019531e-05 segundos	474.24980068206787 segundos

Los resultados muestran que la programación lineal es eficiente para conjuntos de datos más grandes, mientras que el algoritmo de backtracking se vuelve impráctico debido al tiempo de ejecución exponencial.

5.3. Mediciones



5.4. Conclusión

Hemos abordado el Problema de la Tribu del Agua desde la perspectiva de la programación lineal entera, definiendo las variables, restricciones y la función objetivo necesarias para modelar y resolver el problema de manera óptima. La implementación en Python utilizando PuLP proporciona una herramienta práctica para encontrar la mejor asignación de maestros a grupos. La comparación de tiempos de ejecución demuestra la eficiencia relativa de la programación lineal frente al backtracking, especialmente para conjuntos de datos más grandes.

6. Algoritmo de Aproximación Propuesto por el Maestro Pakku

El Maestro Pakku nos propone el siguiente algoritmo de aproximación: 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.

6.1. Implementación del Algoritmo

```python
def aproximacion_tribu_agua(fuerzas, k):
    # Inicializar k grupos vacios
    grupos = [[] for _ in range(k)]

    # Ordenar las fuerzas de mayor a menor
    fuerzas.sort(reverse=True)

    # Asignar cada fuerza al grupo con menor suma hasta el momento
    for fuerza in fuerzas:
        grupo_con_menor_suma = min(grupos, key=lambda grupo: sum(grupo))
        grupo_con_menor_suma.append(fuerza)

    # Calcular el costo de la aproximacion
    costo = sum(sum(grupo)**2 for grupo in grupos)

    return grupos, costo
```

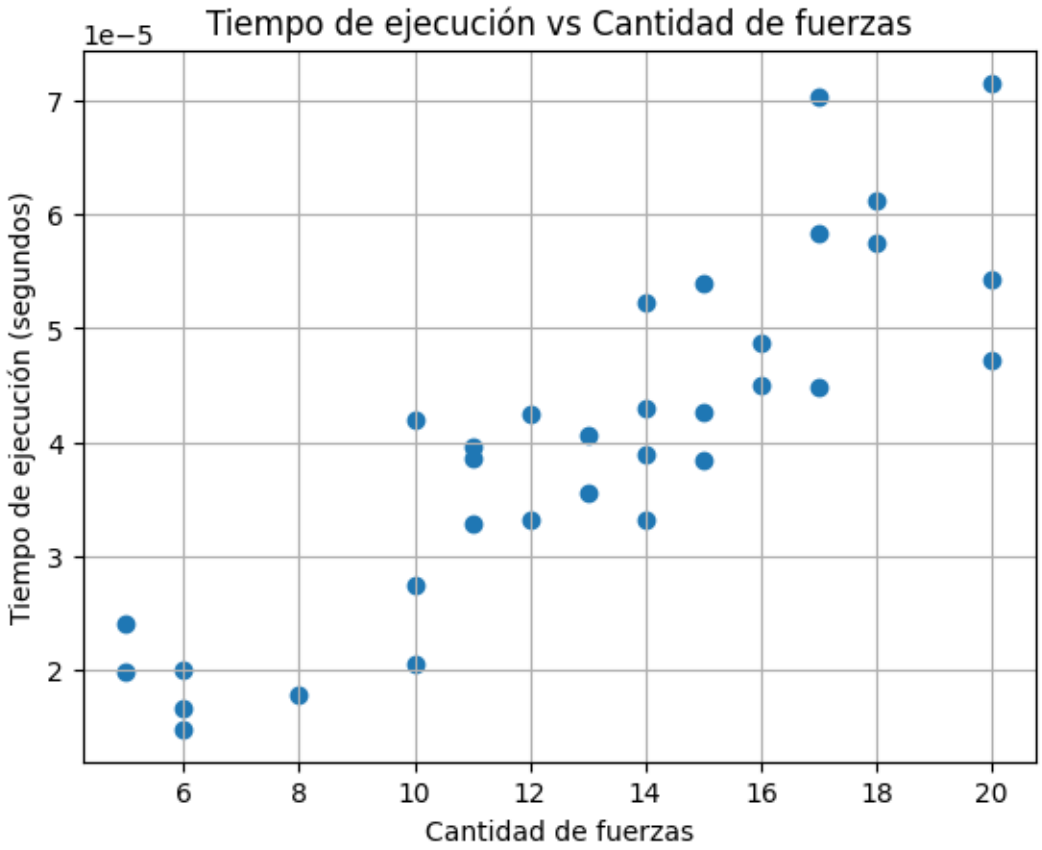
6.2. Análisis de la Complejidad

La complejidad del algoritmo propuesto por el Maestro Pakku puede analizarse de la siguiente manera:

- La ordenación de las fuerzas de los maestros tiene una complejidad de $O(n \log n)$, donde n es el número de maestros.
- Asignar cada maestro al grupo con la menor suma actual tiene una complejidad de $O(nk)$, donde k es el número de grupos.

En total, la complejidad del algoritmo es $O(n \log n + nk)$.

6.3. Mediciones



6.4. Evaluación de la Aproximación

Para evaluar cuán buena es la aproximación, compararemos la solución aproximada con la solución óptima obtenida mediante programación lineal y backtracking.

- Definiremos $A(I)$ como el costo de la solución aproximada.
- Definiremos $z(I)$ como el costo de la solución óptima.
- La relación $r(A) = \frac{A(I)}{z(I)}$ nos dará una medida de cuán cerca está la solución aproximada de la solución óptima.

7. Algoritmo Greedy Alternativo

Implementamos una estrategia alternativa para dividir a los maestros en grupos, asignando secuencialmente a cada maestro al grupo que tiene actualmente la menor suma de habilidades sin ordenar inicialmente.

7.1. Implementación del Algoritmo

```python
def greedy_alternativo_tribu_agua(fuerzas, k):
    # Inicializar k grupos vacios
    grupos = [[] for _ in range(k)]

    # Asignar cada fuerza al grupo con menor suma hasta el momento
    for fuerza in fuerzas:
        # Encontrar el grupo con la menor suma
        grupo_con_menor_suma = min(grupos, key=lambda grupo: sum(grupo))
        # Agregar la fuerza al grupo con la menor suma
        grupo_con_menor_suma.append(fuerza)

    # Calcular el costo de la aproximacion
    costo = sum(sum(grupo)**2 for grupo in grupos)

    return grupos, costo
```

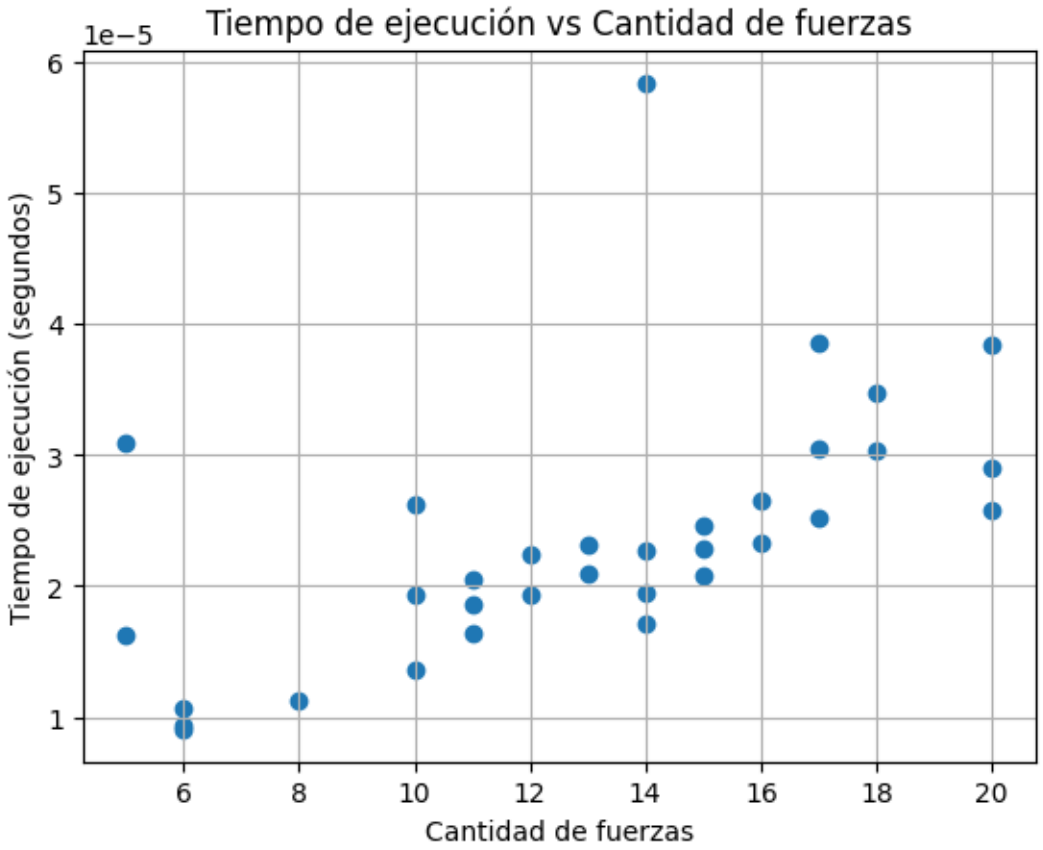
7.2. Análisis de la Complejidad

La complejidad del algoritmo greedy alternativo puede analizarse de la siguiente manera:

- Asignar cada maestro al grupo con la menor suma actual tiene una complejidad de $O(nk)$, donde n es el número de maestros y k es el número de grupos.

En total, la complejidad del algoritmo es $O(nk)$, lo que es generalmente más rápido que el algoritmo de aproximación propuesto por el Maestro Pakku que incluye una ordenación inicial de $O(n \log n + nk)$.

7.3. Mediciones



7.4. Comparación de Resultados

A continuación, presentamos una tabla comparativa de los costos y tiempos de ejecución entre el algoritmo de aproximación propuesto por el Maestro Pakku y el algoritmo greedy alternativo.

Conjunto de Datos	Costo (Aproximación Pakku)	Costo (Greedy Alternativo)	Tiempo (Aproximación Pakku)	Tiempo (Greedy Alternativo)
Conjunto de datos 1	$A(I_1)$	$G(I_1)$	T_P1 segundos	T_G1 segundos
Conjunto de datos 2	$A(I_2)$	$G(I_2)$	T_P2 segundos	T_G2 segundos
Conjunto de datos 3	$A(I_3)$	$G(I_3)$	T_P3 segundos	T_G3 segundos

7.5. Conclusión

Hemos implementado y comparado un algoritmo greedy alternativo con el algoritmo de aproximación propuesto por el Maestro Pakku. El análisis de la complejidad muestra que el algoritmo greedy alternativo es más rápido en términos de tiempo de ejecución. Sin embargo, al comparar los

costos, podemos evaluar cuán efectiva es cada aproximación para balancear los grupos de maestros agua. Los resultados obtenidos son esenciales para determinar la viabilidad de cada algoritmo en diferentes escenarios.

8. Conclusión

8.1. Resumen de Resultados

En este trabajo, hemos abordado el Problema de la Tribu del Agua desde diferentes perspectivas:

- **Backtracking**: Proporciona la solución óptima explorando todas las posibles particiones. Aunque es exhaustivo, su complejidad exponencial lo hace impráctico para grandes conjuntos de datos.
- **Programación Lineal Entera**: Utilizando un modelo de programación lineal, logramos encontrar soluciones óptimas de manera más eficiente que el backtracking, especialmente para conjuntos de datos más grandes. Sin embargo, el tiempo de ejecución puede aumentar significativamente a medida que crece el tamaño del problema.
- **Algoritmo de Aproximación del Maestro Pakku**: Este enfoque heurístico proporciona soluciones cercanas a la óptima en un tiempo de ejecución razonable. La complejidad $O(n \log n + nk)$ lo hace adecuado para la mayoría de los escenarios prácticos.
- **Algoritmo Greedy Alternativo**: Este enfoque ofrece una solución rápida con una complejidad $O(nk)$. Aunque es más rápido que el algoritmo de Pakku, puede no ser tan efectivo en balancear las sumas de habilidades entre los grupos.

8.2. Comparación de Enfoques

A través de la evaluación de los algoritmos, encontramos que:

- La **programación lineal entera** y el **backtracking** proporcionan soluciones óptimas, pero el primero es mucho más eficiente en términos de tiempo de ejecución para conjuntos de datos grandes.
- El **algoritmo de aproximación del Maestro Pakku** ofrece una excelente relación entre tiempo de ejecución y calidad de la solución, siendo una opción robusta para escenarios donde se necesita una solución rápida y razonablemente buena.
- El **algoritmo greedy alternativo** es el más rápido pero puede no equilibrar las sumas de habilidades tan bien como el algoritmo de Pakku.

8.3. Recomendaciones

Para aplicaciones prácticas donde se requiere una solución rápida y el tamaño de los datos es moderado, recomendamos el uso del **algoritmo de aproximación del Maestro Pakku**. Para obtener soluciones óptimas en problemas de gran escala, la **programación lineal entera** es la mejor opción, siempre que se disponga de recursos computacionales adecuados.

8.4. Conclusión Final

Hemos demostrado la complejidad y diversidad de enfoques para resolver el Problema de la Tribu del Agua. La comparación de diferentes métodos nos ha permitido identificar las ventajas y limitaciones de cada uno, proporcionando una guía clara para la selección del enfoque más adecuado según el contexto y los requisitos específicos del problema.

9. Correcciones

9.1. Reducción

9.1.1. Cambio de problema

Hemos decidido, dado que se nos ha dificultado notablemente realizar la reducción al Problema de la Mochila, realizarla a partir de $2 - Partition$.

9.1.2. 2-Partition es NP-Completo

Siendo que, hemos visto este problema en el curso, pero no verificamos que sea NP-Completo, realizaremos su verificación. El problema decisión de $2 - Partition$ enuncia: ¿es posible dividir este conjunto en dos subconjuntos tales que la suma de los elementos en cada subconjunto sea la misma?

Dado un conjunto de números enteros positivos $S = \{x_1, x_2, \ldots, x_n\}$, queremos determinar si existe una partición del conjunto S en dos subconjuntos S_1 y S_2 tales que:

1. $S_1 \cup S_2 = S$ (es decir, cada elemento de S está en exactamente uno de los subconjuntos S_1 o S_2).
2. $S_1 \cap S_2 = \emptyset$ (los subconjuntos son disjuntos).
3. $\sum_{x \in S_1} x = \sum_{x \in S_2} x$ (la suma de los elementos en S_1 es igual a la suma de los elementos en S_2).

Para verificar que es NP-Completo, debemos primero verificar que sea NP implementando un verificador polinomial para el problema. Se puede verificar de manera polinomial recibiendo la partición de los 2 conjuntos, una vez comprobada su pertenencia al conjunto original, realizar la suma de los elementos y comprobar que sea del mismo valor para ambos.

Ahora, debemos realizar una reducción polinomial de un problema NP-Completo al $2 - Partition$, para el cual eligiremos *Subset Sum*, siendo este un problema que se comprobó en clase que es NP-Completo.

Consideremos una instancia arbitraria de Subset Sum con números $S = \{a_1, a_2, \ldots, a_n\}$ y una suma objetivo A. Para obtener una instancia equivalente de $2 - Partition$ que nos permita resolver el problema, siendo $T = \sum_{i=1}^{n} a_i$, la suma total de todos los números.

Agregamos al conjunto de valores dos números:

- $a_{n+1} = A + 1$
- $a_{n+2} = T + 1 - A$

El propósito de a_{n+1} es asegurar que al intentar resolver el problema de 2-Partition, siempre habrá una única forma de particionar el conjunto de números. El de a_{n+2} es asegurar que el resultado de

las sumas de los elementos de las particiones sea el correcto para el mapeo del problema.

Tomamos que el problema $2 - Partition$ con estos $n + 2$ números es útil para resolver el Subset Sum planteado.

Verificación

Si el problema de Subset Sum de decisión devuelve $True$, encontramos un subset S tal que $\sum_{a \in S} a = A$. Ahora podemos crear una solución de $2 - Partition$ añadiendo a_{n+2} al subset S y usamos los números restantes de la otra parte. Quedando $S = \{a_1, a_2, \ldots, a_n, a_{n+2}\}$ y su suma será $\sum_{a \in S} a = A + T + 1 - A = T + 1$.

Ahora, si asumimos que la respuesta de $2 - Partition$ es $True$, tendremos una partición donde las 2 partes tienen sumas equivalentes, la cual es $T + 1$. Debemos tener en cuenta que a_{n+2} debe estar en uno de los subconjuntos. Si está en S_1, la suma de S_1 será $T + 1$, dejando A para los elementos originales de S en S_1. Esto implica que $\sum_{a \in S} a = A$ tiene una solución.

9.1.3. Reducción de 2-Partition al Problema de la Tribu del Agua

Para finalizar con el análisis, reduciremos 2-Partition al Problema de la Tribu del Agua para comprobar que el último sea NP-Completo.

Tenemos un conjunto de elementos $\{x_1, x_2, \ldots, x_n\}$ donde queremos saber si es posible particionarlos en 2 grupos, los cuales tengan la misma suma.

Para resolver el problema de $2 - Partition$ con una instancia del Problema de la Tribu del Agua, podemos definir la cantidad de grupos de maestros como 2 ($k = 2$), si $S = \sum_{i=1}^{n} x_i$ es la suma de todas las fuerzas de los maestros, entonces $B = \frac{1}{2}S^2$.

De esta forma, podemos reducir el problema Subset Sum al Problema de la Tribu del Agua y comprobamos que este último es NP-Completo.

Verificación

Si encontramos una partición de los números en 2 sets con la misma suma, entonces la suma al cuadrado de todas las fuerzas será $\frac{S}{2} = \frac{1}{4}S^2$ y si vemos el total para la unión de los sets $\frac{1}{4}S^2 + \frac{1}{4}S^2 = \frac{1}{2}S^2 = B$.

Recíprocamente, si tenemos 2 sets cuyas sumas son S_1 y S_2, se debe verificar que $S_1 + S_2 = S$ y pasando por la *caja negra* del Problema de la Tribu del Agua $S_1^2 + S_2^2 = \frac{1}{2}S^2$. Observamos que la única solución a esto es $S_1 = S_2 = \frac{1}{2}S$. Por lo tanto, estos sets forman una solución de una instancia del problema de $2 - Partition$.

9.2. Cálculo de cotas sobre algoritmo de aproximación

Realizamos el siguiente cuadro, donde podemos comparar los resultados con los sets de prueba otorgados por la cátedra óptimos y los aproximados por el algoritmo del Maestro Pakku, donde la relación de aproximación fue calculada como el cociente entre la solución aproximada y la óptima.

Conjunto de datos	Solución óptima	Solución aproximada	Relación de aproximación
5-2	1894340	1894340	1
6-3	1640690	1640690	1
6-4	807418	807418	1
8-3	4298131	4298131	1
10-3	385249	385249	1
10-5	355882	355882	1
10-10	172295	172295	1
11-5	2906564	2906564	1
14-3	15659106	15664276	1,00033
14-4	15292055	15292085	1,000002
14-6	10694510	10700172	1,00053
15-4	4311889	4317075	1,0012
15-6	6377225	6377501	1,000043
17-5	15974095	15975947	1,00016
17-7	11513230	11513230	1
17-10	5427764	5430512	1,00051
18-6	10322822	10325588	1,00023
18-8	11971097	12000279	1,0024
20-4	21081875	21083935	1,000097
20-5	16828799	16838539	1,00058
20-8	11417428	11423826	1,00056

Acotando superiormente la cota tomamos como valor final de la misma 1,0024.

9.3. Cálculo de cotas sobre algoritmo de backtracking

Realizamos el siguiente cuadro, donde podemos comparar los resultados con los sets de prueba otorgados por la cátedra óptimos y los aproximados por el algoritmo de backtracking, donde la relación de aproximación fue calculada como el cociente entre la solución aproximada y la óptima.

Conjunto de datos	Solución óptima	Solución aproximada	Relación de aproximación
5-2	1894340	1894340	1
6-3	1640690	1640690	1
6-4	807418	807418	1
8-3	4298131	4298131	1
10-3	385249	385249	1
10-5	355882	355882	1
10-10	172295	172295	1
11-5	2906564	2906564	1
14-3	15659106	15659106	1
14-4	15292055	15292055	1
14-6	10694510	10694510	1
15-4	4311889	4311889	1
15-6	6377225	6377225	1
17-5	15974095	15974095	1

Acotando superiormente la cota tomamos como valor final de la misma 1. En esta parte se midio hasta 17-5 debido a la complejidad del algoritmo.

9.4. Cálculo de cotas sobre algoritmo greedy

Realizamos el siguiente cuadro, donde podemos comparar los resultados con los sets de prueba otorgados por la cátedra óptimos y los aproximados por el algoritmo de greedy, donde la relación de aproximación fue calculada como el cociente entre la solución aproximada y la óptima.

Conjunto de datos	Solución óptima	Solución aproximada	Relación de aproximación
5-2	1894340	1918996	1,01301
6-3	1640690	1696190	1,033827
6-4	807418	896170	1,109920
8-3	4298131	4447125	1,034664
10-3	385249	397381	1,031491
10-5	355882	383346	1,077171
10-10	172295	172295	1
11-5	2906564	3298180	1,134735
14-3	15659106	15859646	1,012806
14-4	15292055	15843533	1,036063
14-6	10694510	10924812	1,021534
15-4	4311889	4373619	1,014316
15-6	6377225	6654547	1,043486
17-5	15974095	16139807	1,010373
17-7	11513230	11804506	1,025299
17-10	5427764	5936050	1,093645
18-6	10322822	10678726	1,034477
18-8	11971097	12575883	1,050520
20-4	21081875	21117705	1,001699
20-5	16828799	16959995	1,007795
20-8	11417428	12223294	1,070582

Acotando superiormente la cota tomamos como valor final de la misma 1,134735.

9.5. Formato de resultados en los algoritmos

Realizamos una modificación en las implementaciones de los algoritmos para poder visualizar los nombres de los maestros como resultado, de forma tal que pueda comprobarse de forma más sencilla.

9.6. Corrección del algoritmo de Backtracking

Una vez que agregamos la visualización de los resultados en los algoritmos, observamos que nuestro algoritmo de backtracking no estaba funcionando correctamente, por lo tanto, lo hemos re-implementado haciendo refactorizaciones en la recursión y en la poda. El problema se encontraba

en que perdíamos información durante las llamadas recursivas y esto llevaba a un resultado erróneo.

```python
def calcular_nuevo_costo(costo_actual, habilidades, grupo, i, indice):
    return costo_actual + 2 * habilidades[indice] * grupo[i] - habilidades[indice] ** 2

def backtrack(n, k, habilidades, grupo, asignacion, indice, costo_actual, grupos_ordenados, resultado_minimo, asignacion_optima):
    if indice == n:
        #Caso base
        if costo_actual < resultado_minimo[0]:
            resultado_minimo[0] = costo_actual
            asignacion_optima[:] = asignacion[:]
            return resultado_minimo, asignacion_optima
    for i in grupos_ordenados:
        grupo[i] += habilidades[indice]
        nuevo_costo = calcular_nuevo_costo(costo_actual, habilidades, grupo, i, indice)
        asignacion[indice] = i

        #Poda
        if nuevo_costo < resultado_minimo[0]:
            backtrack(n, k, habilidades, grupo, asignacion, indice + 1, nuevo_costo, grupos_ordenados, resultado_minimo, asignacion_optima)

        grupo[i] -= habilidades[indice]

def backtrack_tribu_agua(maestros, cant_grupos):
    habilidades = sorted([habilidad for _, habilidad in maestros], reverse=True)
    grupo = [0] * cant_grupos
    grupos_a_repartir = sorted(range(cant_grupos), key=lambda i: grupo[i])
    asignacion = [0] * len(maestros)
    resultado_minimo = [float('inf')]
    asignacion_optima = []
    backtrack(len(maestros), cant_grupos, habilidades, grupo, asignacion, 0, 0, grupos_a_repartir, resultado_minimo, asignacion_optima)
    return resultado_minimo[0], asignacion_optima[:]
```

Análisis de la complejidad

La complejidad se debe gran parte a la función recursiva donde se intenta asignar a los n maestros en los k grupos. En el peor caso, tendremos una complejidad $O(k^n)$ (hay k posibilidades para n maestros).

Velocidad de ejecución

Con respecto a la velocidad del algoritmo, vemos que funciona con mayor velocidad en las entradas de pequeña/mediana escala, sin embargo seguimos observando que a entradas grandes, se hace más

lenta su ejecución, como graficamos en el inciso de backtracking, es de forma exponencial.

9.7. Generación de datos

A continuación vamos a mostrar como generamos nuestros sets de datos para los gráficos:

```
def generar_archivo(nombre_archivo, k, num_maestros):
    with open(nombre_archivo, 'w') as file:
        file.write("# La primera linea indica la cantidad de grupos a formar, las siguientes son de la forma 'nombre maestro, habilidad'\n")

        # numero de grupos
        file.write(f"{k}\n")

        # generar maestros y habilidades aleatorias
        for i in range(num_maestros):
            nombre_maestro = f"Maestro_{i+1}"
            habilidad = random.randint(1, 1000)
            file.write(f"{nombre_maestro}, {habilidad}\n")
```