.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

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

17 de noviembre de 2025

Agustin Bermudez	Mateo Gonzalez Pautaso	Tiago Calderón
111863	111699	111894

1. Demostración que el Problema de la Tribu del Agua se encuentra en NP.

Primero, se plantea el problema de decisión correspondiente: dado un conjunto de maestros y un número k, ¿es posible separarlos en k subgrupos tales que la suma de los cuadrados de las fuerzas totales de cada grupo sea, como máximo, un valor B?

Para demostrar que este problema está en NP, construimos un validador para verificar si la solución es correcta. Este validador tiene que funcionar en tiempo polinomial.

Parámetros del verificador

- **maestros_agua:** Representa el set de datos representando a los diferentes maestros, siendo un diccionario con el formato nombre: poder
- **k:** número entero que representa la cantidad de subgrupos en los que deben ser separados los maestros
- **B:** Es el valor máximo que puede alcanzar la adición de los cuadrados de las sumas de las fuerzas de los grupos.
- **subgrupos:** Es la solución a validar. Consiste en una lista de listas, cada subgrupo es una lista de nombres de maestros

Dicho algoritmo se puede observar en la siguiente sección de código:

```python
def grupos_parejos(maestros_agua, k, B, subgrupos):
    maestros_dict = {nombre: poder for nombre, poder in maestros_agua}
    maestros_agua = maestros_dict

    if len(subgrupos) != k:
        return False    # La cantidad de subgrupos no coincide con k

    asignados = set()
    B_conseguido = 0

    for subgrupo in subgrupos:
        poder_subgrupo = 0

        for nombre_maestro in subgrupo:

            if nombre_maestro in asignados or nombre_maestro not in maestros_agua:
                return False    # El maestro solo puede estar asignado a un
    subgrupo

            asignados.add(nombre_maestro)
            poder_subgrupo += maestros_agua[nombre_maestro]

        B_conseguido += poder_subgrupo ** 2

    if len(asignados) != len(maestros_agua):
        return False    # Cada maestro debe estar asignado a un grupo

    return B_conseguido <= B
```

Complejidad del Validador:

1. Lo primero que se hace es convertir la lista de tuplas a un diccionario y luego verificar si la longitud de la solución recibida, es decir, la cantidad de subgrupos en los que se separaron los maestros, es igual al valor k pasado por parámetro: O(n) + O(1)
2. A continuación, se procederá a revisar los subgrupos y, dentro de estos, se examinarán los maestros presentes, verificando la validez de las asignaciones y calculando el valor B de la solución obtenida. Como cada maestro debe estar asignado a un solo grupo, la cantidad de iteraciones que se van a realizar son O(n + k), siendo n la cantidad de maestros.

3. Las verificaciones siguientes en el codigo no aportan complejidad adicional, ya que son O(1)

Por lo tanto, el comportamiento del validador de la solución para el problema de Maestros del Agua, es polinomial, concluyendo que el problema esta en NP.

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

Para demostrar que el problema enunciado es, en efecto, NP-Completo, debemos reducir un problema NP-Completo a este. Para esta reducción elegimos reducir Partition-Problem a Tribu Del agua, lo que implica resolver Partition-Problem con un algoritmo que resuelva Tribu del Agua

2.1. Demostración que Partition-Problem es NP-Completo

Para poder demostrar que Partition-Problem es un problema NP-Completo, vamos a reducir un problema que ya sabemos que esta dentro de esta categoría al mismo. El problema elegido para esto es Subset-Sum

Primero se plantea el problema de decisión de Partition-Problem: Dado un conjunto X de n elementos, ¿Se puede dividir X en dos subconjuntos de igual suma?

2.1.1. Pertenencia a NP

A continuación se muestra un validador para la solución que funciona en tiempo polinomial, demostrando que este problema pertenece a NP.

```
def partition_problem_valido(X, asignaciones):
    '''
    X: lista de elementos
    asignaciones: lista de 0s y 1s del mismo largo que X, donde 1 indica que el elemento va al subconjunto S1 y 0 que va a S2.
    '''

    if len(X) != len(asignaciones):
        return False

    suma_s1 = 0
    suma_s2 = 0

    for i in range(asignaciones):
        asig = asignaciones[i]
        if asig == 1:
            suma_s1 += X[i]
        elif asig == 0:
            suma_s2 += X[i]
        else:
            return False

    return suma_s1 == suma_s2
```

2.1.2. Reducción de Subset-Sum a Partition-Problem

Para lograr esta reducción, se debe poder resolver el problema de Subset-Sum usando un algoritmo que pueda resolver Partition-Problem, mediante una cantidad de pasos polinomial en el medio. Para lograr esto, vamos a transformar el problema:

Los parámetros que va a recibir Subset-Sum son los siguientes:

- X: como un conjunto de elementos $\{x_1, x_2, ..., x_n\}$

- t: un número objetivo

Estos parámetros se transformarán en los siguientes para enviárselos al algoritmo que resuelve Partition-Problem:

- Primero, defino $\sigma = \sum_{xi \in X} x_i$
- $X' = X \cup \{2\sigma - t, \sigma + t\} = \{x_1, x_2, ..., x_n, 2\sigma - t, \sigma + t\}$
- La suma de todos los elementos de X' equivale a 4σ
- Siguiente, resuelvo Partition-Problem, pasándole como parámetro a X'

Ahora se demuestra la ida y la vuelta de la reducción

1. **Ida: Cuando hay subset-sum hay partition**

   Como hay subset-sum, existe el Subconjunto $S \subseteq X$ cuya suma de elementos es el valor objetivo t.
   Por lo tanto, la suma de los elementos fuera de S es $\sigma - t$.
   Ahora considero el subconjunto de X' llamado $A = S \cup \{2\sigma - t\}$. La suma de sus elementos es $t + (2\sigma - t) = 2\sigma$.
   El complemento de A en X' es $B = (X' - A)$, que es lo mismo que: $B = (X - S) \cup \{\sigma + t\}$. La suma del subconjunto B es lo que sobra de X y el otro término agregado: $(\sigma - t) + (\sigma + t) = 2\sigma$.
   Como A y su complemento B suman lo mismo, el conjunto X' es particionable. Por lo tanto, si existe solución al Subset-Sum, existe solución a Partition-Problem

2. **Vuelta: Cuando hay partition hay subset-sum**

   X' debería poder particionarse en dos subconjuntos de igual suma 2σ.
   En la subdivisión los elementos $(2\sigma - t)$ y $(\sigma + t)$ no pueden estar en la misma partición, pues su suma equivale a 3σ, que excede la mitad. Por lo tanto, uno esta sumándose a un subconjunto A y otro a un subconjunto B.
   Defino los valores:

   - $a = \sum_{ai \in A} a_i$
   - $b = \sum_{bi \in B} b_i$
   - sabiendo que $a + b = \sigma$

$$a + (\sigma + t) = b + (2\sigma - t) \tag{1}$$
$$a - b = \sigma - 2t \tag{2}$$
$$a - (\sigma - a) = \sigma - 2t \tag{3}$$
$$2a - \sigma = \sigma - 2t \tag{4}$$
$$2a = 2\sigma - 2t \tag{5}$$
$$a = \sigma - t \tag{6}$$

   Por lo que b sería lo mismo que $\sigma - t + b = \sigma$ y esto daría como resultado $b = t$, lo que representa que B es el subconjunto solución del problema del Subset-Sum. Concluyendo con esto, la existencia del Partition-Problem también garantiza la existencia del Subset-Sum

2.1.3. Conclusión

Finalmente, habiendo demostrado que Partition-Problem es un problema que está en NP, y habiendo reducido un problema NP-Completo conocido (visto en clase) como Subset-Sum a Partition-Problem, se concluye que este último también es un problema NP-Completo.

2.2. Reducción de Partition-Problem a Tribu del Agua

Previamente, se plantearon el verificador en tiempo polinómico y el problema de decisión correspondiente a ambos casos. A continuación, se muestra la reducción:

$$\textit{Partition-Problem} \leq_{\text{p}} \textit{Tribu del Agua}$$

Para ello, debemos ser capaces de resolver una instancia de *Partition-Problem* utilizando un algoritmo que resuelva *Tribu del Agua*, empleando una cantidad de pasos polinómica.

Los parámetros que recibe *Partition-Problem* son los siguientes:

- $X = \{x_1, x_2, ..., x_n\}$

Estos deben transformarse de manera adecuada para que el algoritmo de *Tribu del Agua* pueda resolverlo:

- $\tau = \sum_{x_i \in X} x_i$
- $X' = X = \{x_1, x_2, ..., x_n\}$
- $k = 2$
- $B = 2(\frac{1}{2}\tau)^2 = \frac{1}{2}\tau^2$

Ahora se demuestra la ida y la vuelta de la reducción

1. **Ida: Cuando hay Partition, hay Tribu**

   Supongamos que hay Partition, entonces existen dos subconjuntos $L \subseteq X$ y $R = X - L$ tal que $\sum_{l_i \in L} l_i = \sum_{r_i \in R} r_i = \frac{1}{2}\tau$.
   Donde la suma de los cuadrados de ambos subconjuntos es:

   $$(\sum_{l_i \in L} l_i)^2 + (\sum_{r_i \in R} r_i)^2 = (\tfrac{1}{2}\tau)^2 + (\tfrac{1}{2}\tau)^2 = 2(\tfrac{1}{2}\tau)^2 = B$$

   Por lo tanto, la instancia de Tribu del Agua se cumple.

2. **Vuelta: Cuando hay Tribu, hay Partition**

   Supongamos que hay Tribu y admite la partición en dos subgrupos L y R tal que

   $$S_L = \sum_{l_i \in L} l_i \qquad S_R = \sum_{r_i \in R} r_i \qquad S_L^2 + S_R^2 \leq B$$

   y además $S_L + S_R = \tau$. Si expandimos el cuadrado de la suma

   $$(S_L + S_R)^2 = S_L^2 + S_R^2 + 2S_L S_R = \tau^2$$

   De ahí se obtiene:

   $$S_L^2 + S_R^2 = (S_L + S_R)^2 - 2S_L S_R = \tau^2 - 2S_L S_R \leq B$$

   Luego:

   $$\tau^2 - 2S_L S_R \leq \tfrac{1}{2}\tau^2 \longrightarrow -2S_L S_R \leq -\tfrac{1}{2}\tau^2 \longrightarrow S_L S_R \leq \tfrac{1}{4}\tau^2$$

   El producto $S_L S_R$ alcanza su máximo $\frac{1}{4}\tau^2$ únicamente cuando $S_L = S_R = \frac{1}{2}\tau$
   Por lo tanto, la existencia de una solución para *Tribu del Agua* implica la existencia de una partición de sumas iguales para *Partition-Problem*

2.2.1. Conclusión

Hemos mostrado que:

- *Tribu del Agua* tiene un verificador en tiempo polinómico (se demostró previamente), por lo tanto pertenece a NP.
- Existe una reducción en tiempo polinómico *Partition-Problem* $\leq_{\mathrm{p}}$ *Tribu del Agua* cuya corrección quedó probada en ambas direcciones.

Por lo tanto, dado que *Partition-Problem* es NP-Completo y existe una reducción polinómica desde él hacia *Tribu del Agua*, se concluye que *Tribu del Agua* también es NP-Completo.

3. Algoritmo Backtracking

Para hallar la solución óptima al problema de la Tribu del Agua, se creó un algoritmo de backtracking que investiga todas las asignaciones posibles de los maestros en cada uno de los k grupos, con el objetivo de encontrar aquella asignación que minimice el coeficiente posible, es decir, la suma de los cuadrados de los poderes.

El algoritmo usado para esto es el siguiente:

```
def _tribu_del_agua_bt(maestros_agua, k, i, sol_actual, adic_actual,
    suma_grupos_actual, mejor_sol, adic_mejor, poder_restante):
    if i >= len(maestros_agua):
        if adic_actual < adic_mejor:
            copia = [list(g) for g in sol_actual]
            return copia, adic_actual
        return mejor_sol, adic_mejor

    maestro = maestros_agua[i]
    nombre, poder = maestro
    restante = poder_restante[i + 1]

    for g in range(k):
        suma_sin = suma_grupos_actual[g]
        suma_con = suma_sin + poder
        adic_con = adic_actual - (suma_sin ** 2) + (suma_con ** 2)

        sol_actual[g].append(maestro)
        suma_grupos_actual[g] = suma_con

        cota_minima = calcular_cota_minima(suma_grupos_actual, restante, k)
        if cota_minima >= adic_mejor:
            sol_actual[g].pop()
            suma_grupos_actual[g] = suma_sin
            continue

        sol_res, adic_res = _tribu_del_agua_bt(maestros_agua, k, i + 1, sol_actual,
     adic_con, suma_grupos_actual, mejor_sol, adic_mejor, poder_restante)

        if adic_res < adic_mejor:
            mejor_sol, adic_mejor = sol_res, adic_res

        sol_actual[g].pop()
        suma_grupos_actual[g] = suma_sin

        if suma_sin == 0:
            break

    return mejor_sol, adic_mejor

def tribu_del_agua_bt(maestros_agua, k):
    '''
    maestros_agua = es una lista de tuplas de la forma [(nombre, poder), ...]
    k = la cantidad de subgrupos a armar
    '''
    sol_actual = [[] for i in range(k)]
    suma_grupos_actual = [0] * k

    mejor_sol, adic_mejor = aproximador_pakku(maestros_agua, k)

    maestros_agua = sorted(maestros_agua, key=lambda x:x[1], reverse=True)

    n = len(maestros_agua)
    poder_restante = [0] * (n + 1)
    for i in range(n - 1, -1, -1):
        poder_restante[i] = poder_restante[i + 1] + maestros_agua[i][1]

    return _tribu_del_agua_bt(maestros_agua, k, 0, sol_actual, 0,
    suma_grupos_actual, mejor_sol, adic_mejor, poder_restante)
```

3.1. Podas y mejoras al algoritmo

En las primeras iteraciones del algoritmo, no se estaban implementando suficientes optimizaciones como para lograr ejecutar los datasets provistos por la cátedra en un tiempo razonable. Por esta razón, se decidió implementar diversas podas y heurísticas para minimizar al máximo posible la demora del algoritmo al momento de ejecutarse con datasets de tamaño considerable. A continuación se detalla una explicación de cada una de las mejoras implementadas en el algoritmo:

1. **Heurística: ordenar a los maestros por poder decreciente**

   Al ordenar a los maestros de esta forma, el algoritmo prioriza el procesamiento de los maestros de mayor poder. De esta manera, se toman las decisiones más impactantes en las llamadas más tempranas de la recursión. Esto tiene dos grandes beneficios: Primero aumenta la probabilidad de encontrar una solución de buena calidad más rápido

   Segundo, hace que la cota mínima crezca más rápido, logrando así que las podas sean efectivas de manera más temprana.

2. **Optimización: guardar un arreglo con el poder restante a asignar para cada iteración**

   Para ahorrar el cálculo en cada llamada recursiva del poder restante a asignar, se optó por pre-calcular un arreglo que guarde estos valores y los pase como una constante a la función recursiva `_tribu_del_agua_bt`.

   Este valor es fundamental para el cálculo de la poda por cota mínima. Con esta optimización de pre-cálculo, en lugar de hacer un cálculo $O(n)$ en cada llamada recursiva, se hace un solo recorrido $O(n)$ una única vez para calcular el arreglo `poder_restante` y luego realizar accesos $O(1)$ en la recursión, logrando así optimizar el tiempo.

3. **Optimización: Inicializar la mejor solución encontrada con el aproximador de Pakku**

   En las primeras versiones de este algoritmo de backtracking, la mejor solución se inicializaba vacía. Esto provocaba que la poda por cota mínima no funcionara hasta que se encontraba la primera asignación válida, pudiendo ser de mala calidad.

   Para evitar estos problemas, `adic_mejor` se inicializa con el aproximador Greedy planteado en el apartado 5, logrando que la poda por cota mínima funcione desde la primera llamada. Aunque el cálculo de la solución aproximada represente una complejidad extra de $O(n^2)$, ahorra mucho mas tiempo al accionar las podas que lo que demora en llegar a la solución aproximada.

4. **Poda: Calcular una cota mínima para cada estado de la recursión**

   Esta poda es la más importante del algoritmo y esta implementada en el siguiente bloque de código:

```
        ...

        cota_minima = calcular_cota_minima(suma_grupos_actual, restante, k)
        if cota_minima >= adic_mejor:
            sol_actual[g].pop()
            suma_grupos_actual[g] = suma_sin
            continue

        ...

```

   Funciona de la siguiente manera: dado un estado actual de asignaciones de maestros y un cierto poder restante, la funcion `calcular_cota_minima` va a calcular el mínimo coeficiente teórico que se puede alcanzar con ese estado actual. El calculo lo hace de la siguiente manera:

   a) Desde un arreglo que representa la sumatoria de poder de cada grupo, va a crear una copia y ordenarlos ascendentemente.

b) Iterativamente, va a ir equiparando el poder de los grupos más débiles hasta el nivel del siguiente más fuerte. La lógica entonces es, aumentar el poder el grupo 0 para que alcance al grupo 1; luego aumentar los poderes de los grupos 0 y 1 para que alcancen al 2, y así sucesivamente.

Si se da el caso donde no alcanza el poder para equiparar los grupos, reparte el poder restante equitativamente entre los grupos que se estaban intentando nivelar

c) En el caso donde ya están todos los grupos equiparados con el mismo nivel de poder, pero aún sobra poder restante, lo reparte equitativamente entre los k grupos.

d) Luego calcula la adición de los cuadrados de los poderes de cada grupo presente en el arreglo.

e) Este valor va a representar el mínimo coeficiente teórico que podría alcanzar el estado actual. Este valor siempre será menor o igual al real, porque en la realidad no se puede dividir el poder como convenga ya que este depende de los cada maestro.

Con el valor de la cota mínima, se compara con el coeficiente de la mejor solución ya encontrada. En caso de que esta cota mínima `cota_minima` sea mayor o igual que la mejor solución encontrada `adic_mejor`, no hay manera posible de que el estado actual llegue a ser mejor, por lo tanto, se poda esa rama.

5. **Poda: Evitar asignaciones equivalentes para los grupos vacíos**

   Esta poda evita explorar asignaciones de los grupos que son permutaciones equivalentes presentes por ubicar a un maestro en un grupo vacío.

   Por ejemplo, con $k = 3$ grupos vacíos, asignar al primer maestro al grupo 1,2 o 3 produce resultados que son idénticos. Esta poda esta implementada en las lineas 34 y 35 del algoritmo.

3.2. Pruebas para la correctitud del algoritmo

Para corroborar que el algoritmo estaba efectivamente llegando a calcular los resultados óptimos, creamos un archivo que almacena una prueba por cada uno de los resultados conocidos de los datasets de la cátedra.

Para estas pruebas se usaron funciones preparadas para ejecutar el archivo de pruebas `test_catedra.py` usando `pytest`. Estas pruebas se encargan de verificar que el coeficiente alcanzado por el algoritmo de backtracking es efectivamente el mismo que el conocido. La forma de las pruebas es la siguiente:

```
    def test_5_2():
        archivo = '5_2.txt'
        path_set = f"./datasets/{archivo}"
        x, f = parser(path_set)
        _, resultado = tribu_del_agua_bt(x, f)
        assert resultado == 1894340
        print(f"Coeficiente de {archivo}: {resultado}")
```

Para ejecutar el set de pruebas se debe utilizar el siguiente comando:

```
pytest ./test_catedra.py
```

Al ejecutar el comando se puede observar que el resultado de cada uno de las datasets coincide con el esperado.

```
pytest ./test_catedra.py
========================== test session starts ===========================
platform linux -- Python 3.12.3, pytest-7.4.4, pluggy-1.4.0
rootdir: /home/mateo-gonzalez-pautaso/Desktop/TB024-TdA-BCGP
collected 21 items

test_catedra.py .....................                                  [100%]

============================ 21 passed in 3.72s ============================
```

3.3. Mediciones de tiempos

El algoritmo de backtracking planteado en este apartado tiene una complejidad temporal exponencial. Si bien las diversas podas y heurísticas implementadas para reducir el espacio de búsqueda ayudan a minimizar los tiempos de ejecución, no cambian la naturaleza exponencial del algoritmo.

A continuación se muestran ciertas mediciones de los tiempos de ejecución del algoritmo, ajustando la curva de tiempo a la función cuadrática y a la función exponencial, con el fin de dimensionar el impacto de la longitud de los dataset en relación al tiempo de ejecución.

3.3.1. Análisis de rendimiento para el caso $k = 4$

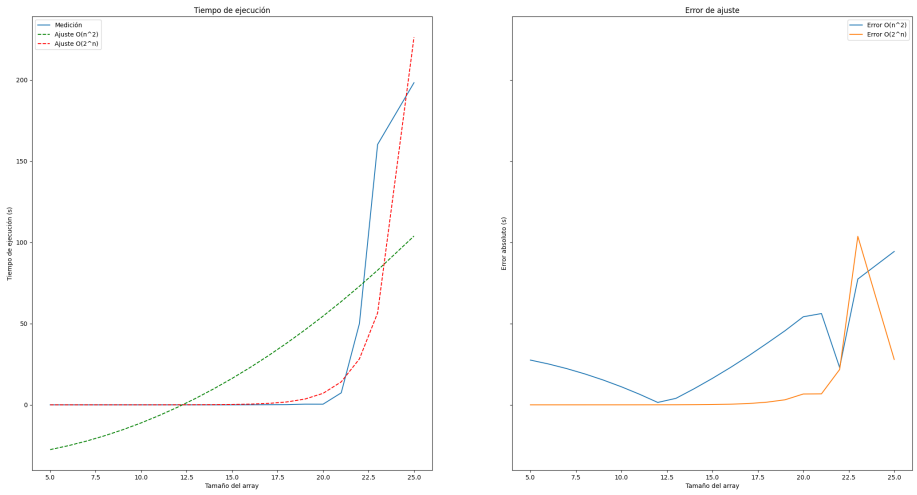

Figura 1: Resultados con *poder* $\in [1, 500]$ y $k = 4$

3.3.2. Análisis de rendimiento para el caso $k = 6$

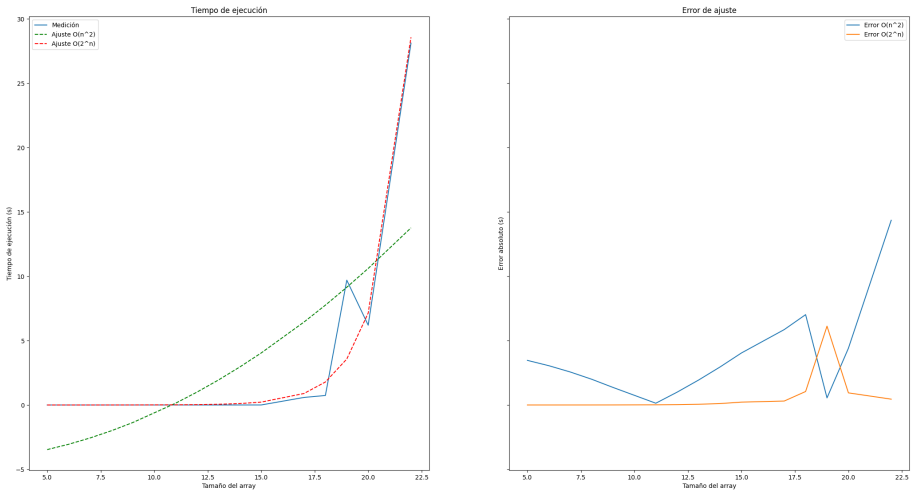

Figura 2: Resultados con *poder* $\in [1, 500]$ y $k = 6$

3.3.3. Análisis de rendimiento para el caso $k = 8$

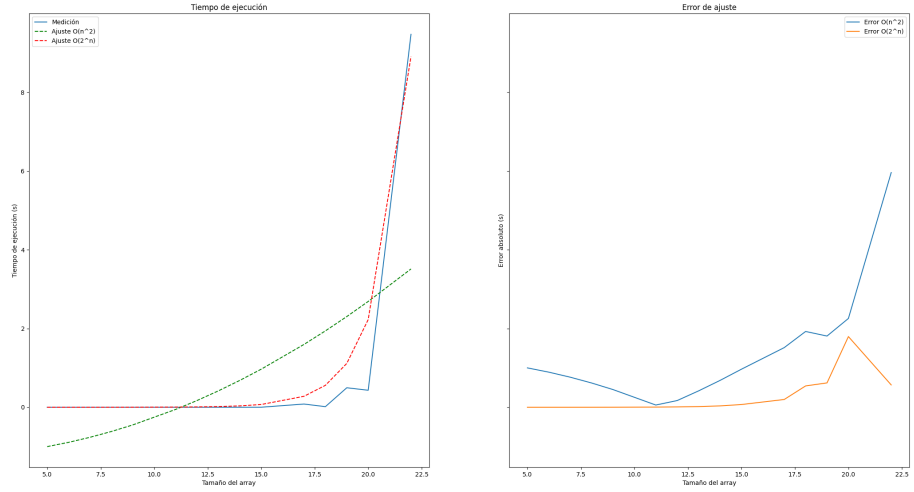

Figura 3: Resultados con *poder* $\in [1, 500]$ y $k = 8$

4. Algoritmo Programación Lineal

4.1. Implementación

Se propone un algoritmo que utiliza programación lineal para resolver aproximadamente el problema en el que se basa este trabajo práctico:

```python
def tribu_del_agua_pl(maestros_agua, k):
    maestros_agua = sorted(maestros_agua, key=lambda x: x[1], reverse=True)
    n = len(maestros_agua)

    nombres = {j: maestros_agua[j-1][0] for j in range(1, n+1)}
    fuerzas = {j: maestros_agua[j-1][1] for j in range(1, n+1)}

    prob = LpProblem("tribu_del_agua", LpMinimize)

    A = LpVariable.dicts("A",
                         (range(1, k+1), range(1, n+1)),
                         cat=LpBinary)

    Z = LpVariable("Z", lowBound=0)
    Y = LpVariable("Y", lowBound=0)

    prob += Z - Y

    for j in range(1, n+1):
        prob += lpSum(A[i][j] for i in range(1, k+1)) == 1

    S = {
        i: lpSum(A[i][j] * fuerzas[j] for j in range(1, n+1))
        for i in range(1, k+1)
    }

    for i in range(1, k+1):
        prob += Z >= S[i]
        prob += Y <= S[i]

    top_k = min(k, n)
    for idx in range(1, top_k+1):
        prob += A[idx][idx] == 1

    prob.solve(PULP_CBC_CMD(msg=0))
    return reconstruir(k, S, nombres, fuerzas, A)

def reconstruir(k, S, nombres, fuerzas, A):
    asign = [[] for _ in range(k)]
    suma_cuad = 0

    for i in range(1, k+1):
        si = value(S[i])
        suma_cuad += si * si
        for j in range(1, len(nombres)+1):
            if value(A[i][j]) > 0.5:
                asign[i-1].append((nombres[j], fuerzas[j]))

    return asign, int(suma_cuad)
```

4.2. Modelo de Programación Lineal

Parámetros

$$f_j = \text{fuerza del maestro } j$$

Variables

$$A_{ij} = \begin{cases} 1, & \text{si el maestro } j \text{ pertenece al grupo } i, \\ 0, & \text{en caso contrario} \end{cases} \qquad \text{(variable binaria)}$$

$$Z = \text{máximo entre las sumas de fuerzas de los grupos}$$

$$Y = \text{mínimo entre las sumas de fuerzas de los grupos}$$

$$S_i = \sum_{j=1}^{n} A_{ij}\, f_j \qquad \text{(suma de fuerzas del grupo } i\text{)}$$

Función objetivo

$$\text{mín } Z - Y$$

Restricciones

- **Asignación exacta: cada maestro debe pertenecer a un único grupo**

$$\sum_{i=1}^{k} A_{ij} = 1 \qquad \forall j$$

- **Cotas inferiores**

$$Z \geq 0, \qquad Y \geq 0$$

- **Definición de suma máxima**

$$Z \geq S_i \qquad \forall i$$

- **Definición de suma mínima**

$$Y \leq S_i \qquad \forall i$$

4.3. Detalles del algoritmo

- Se ordenan los maestros de mayor a menor fuerza para optimizar los tiempos de ejecución del algoritmo(acelera las podas del árbol en Branch and Bounds)
- A la hora de crear ciertas variables se utiliza $lowBound = 0$ debido a que esto nos evita poner la restricción de que dichas variables deben ser ¿= 0
- Se resuelve el problema con el solver CBC forzando que no se muestre ningún mensaje por pantalla con msg = 0
- Con `LpVariable.dicts` se crean variables de decisión a la vez, guardadas en un diccionario indexado.
- Con el siguiente fragmento de código, se coloca a los min(k, n) maestros más fuertes en grupos distintos. Aunque esto puede empeorar la cota de aproximación o la optimalidad del algoritmo, mejora enormemente sus tiempos de ejecución. Por esto que mencionamos decidimos utilizarlo

```
1                top_k = min(k, n)
2                for idx in range(1, top_k+1):
3                    prob += A[idx][idx] == 1
4
```

4.4. Cota de aproximación empírica

Para encontrar la cota de aproximación empírica de este modelo de Programación Lineal (que optimiza el rango), se van a ejecutar todos los datasets provistos por la cátedra y se analizará la diferencia entre el óptimo y el resultado conocido.

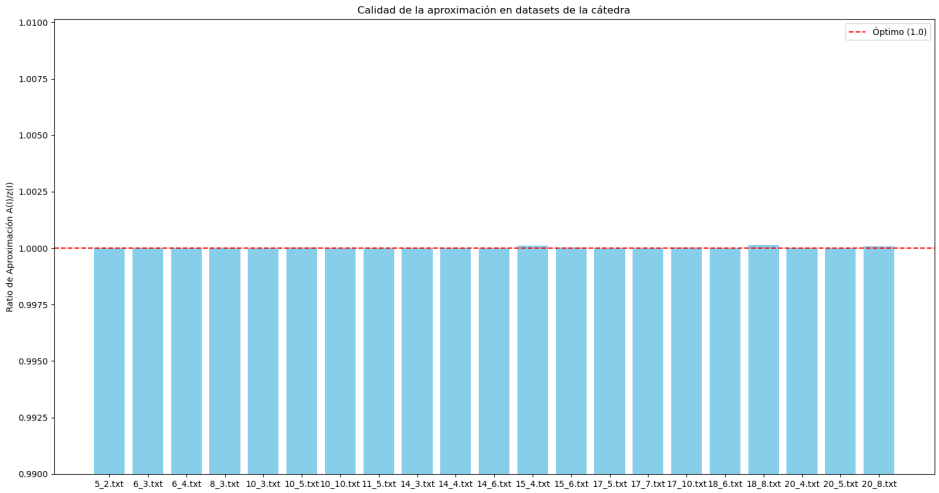

Figura 4: Resultados de $r(I)$ para datasets de la cátedra

Cómo se puede apreciar en la imagen, la diferencia entre el resultado óptimo y el resultado que logra aproximar la implementación mostrada en programación lineal es mínima, logrando valores para $r(I)$ menores a 1.0025 en todos los casos. Concluyendo así que el aproximador por programación lineal consigue muy buenas aproximaciones.

4.5. Mediciones de tiempo y comparación con backtracking

Las mediciones de tiempo se realizarán comparando los tiempos del algoritmo implementado por programación lineal con el que fue realizado utilizando Backtracking. Esto se verá reflejado en un gráfico de barras donde se podrán observar los tiempos de estos algoritmos.

4.5.1. Ajuste a la función exponencial

A continuación se muestra el gráfico de ajuste a la función exponencial para demostrar que el algoritmo de programación lineal con variables enteras mostrado en este apartado es efectivamente de complejidad exponencial

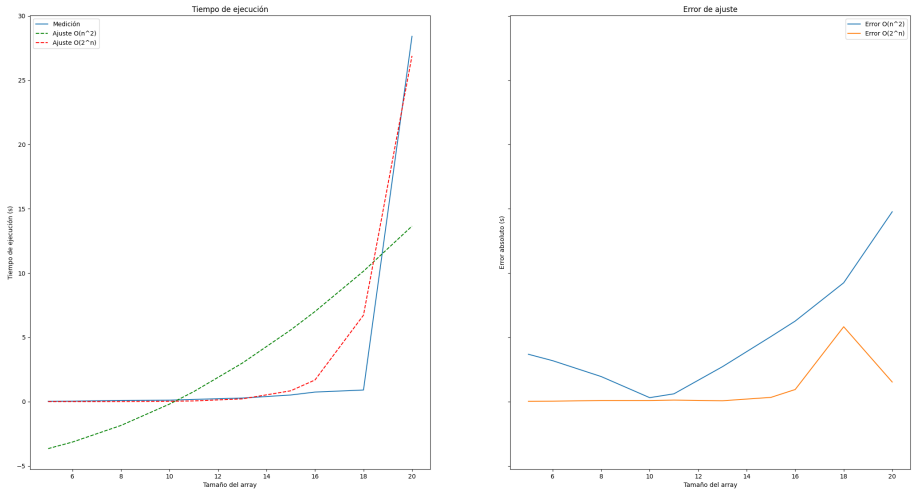

Figura 5: Gráfico de ajuste de programación lineal

4.5.2. Comparación contra backtracking

Para estudiar la diferencia de los tiempos de ejecución entre este algoritmo y el implementado con backtracking, se realizaron gráficos de barras donde se modela los diferentes tiempos que tardó cada algoritmo en algunos de los datasets relevantes provistos por la cátedra

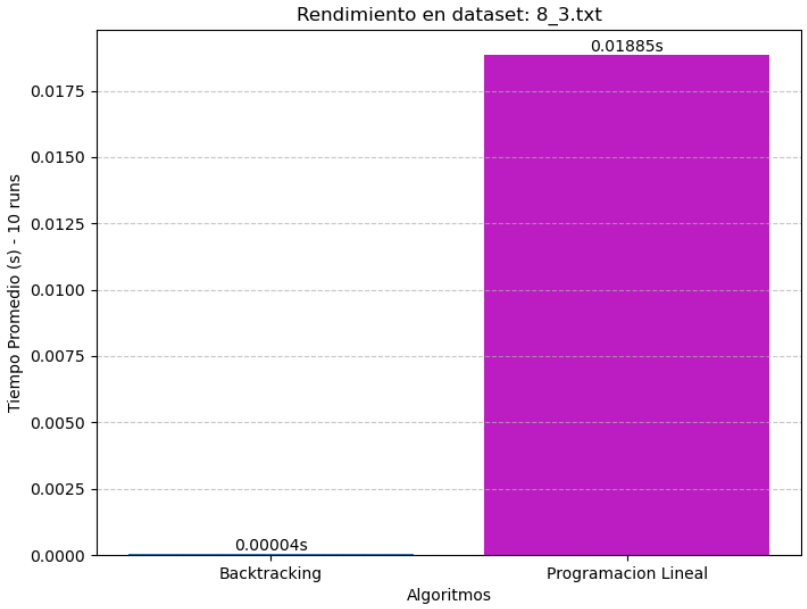

Figura 6: Comparación Programación Lineal vs Backtracking en 8_3.*txt*

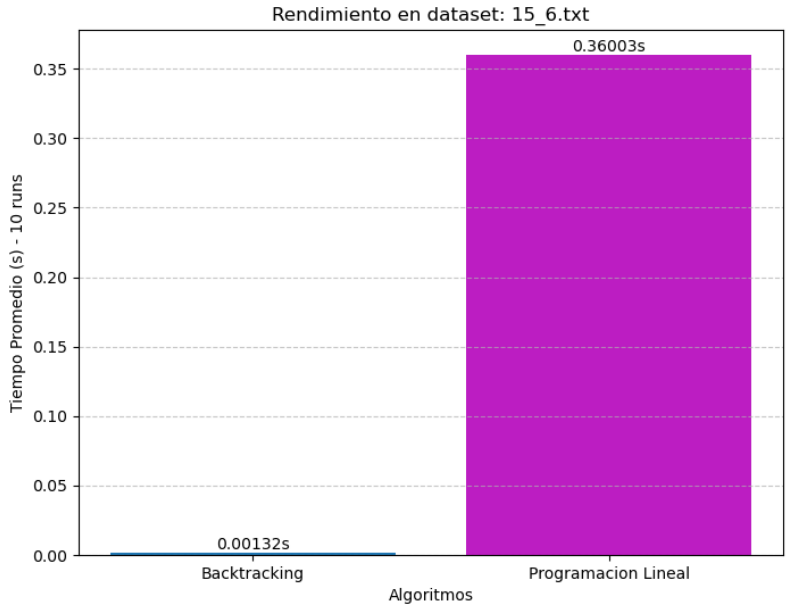

Figura 7: Comparación Programación Lineal vs Backtracking en 15_6.*txt*

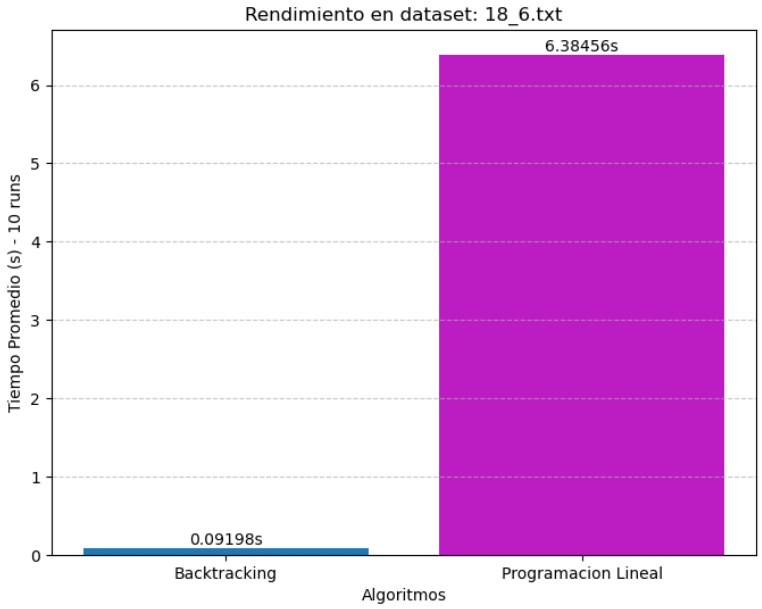

Figura 8: Comparación Programación Lineal vs Backtracking en 18_6.*txt*

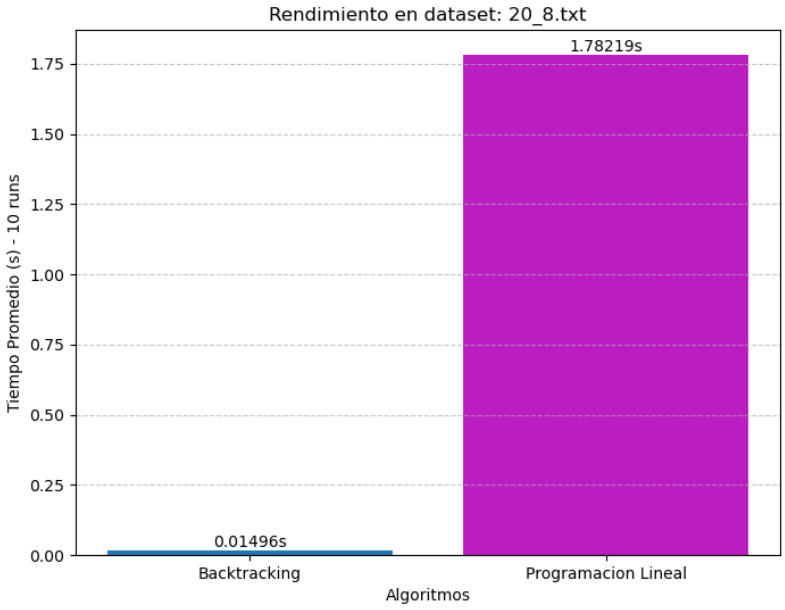

Figura 9: Comparación Programación Lineal vs Backtracking en 20_8.*txt*

Como se puede observar, es muy clara la diferencia en tiempos de ejecución entre ambos algoritmos. Mediante los datasets provistos por la cátedra, donde Backtracking no tarda mas de 1 segundo, Programación Lineal se dispara a 100 veces más. Es muy evidente

5. Aproximador de Pakku

5.1. Implementación y detalles del algoritmo

Parámetros del aproximador

- **maestros_agua:** Representa el set de datos representando a los diferentes maestros, siendo una lista de tuplas (maestro, poder)
- **k:** número entero que representa la cantidad de subgrupos en los que deben ser separados los maestros

El algoritmo implementa una estrategia Greedy para intentar minimizar la suma de los cuadrados de las fuerzas de los grupos. Antes de distribuir a los maestros, es fundamental ordenarlos de forma descendente según su poder; así nos aseguramos de que los maestros con mayor poder sean asignados primero a los grupos vacíos o con menor suma. Utilizamos a los maestros con menor poder para nivelar al final.

Para mantener una implementación eficiente, se utilizan dos técnicas en el manejo de los datos:

- **Arreglo auxiliar de sumas:** Se mantiene una lista auxiliar `suma_grupos_pakku` que almacena la carga actual de cada grupo. Esto evita tener que recorrer y sumar las listas de maestros dentro de cada grupo repetidamente, reduciendo el costo de calcular la suma total en cada paso.
- **Cálculo incremental del costo:** En lugar de recalcular la suma de los cuadrados desde cero en cada asignación, se realiza una actualización diferencial. Se resta el cuadrado de la suma anterior del grupo modificado y se suma el cuadrado de la nueva suma:

$$adic\_pakku = adic\_pakku - (suma\_sin^2) + (suma\_con^2)$$

Dicho algoritmo se puede observar en la siguiente sección de código:

```python
def grupo_min(suma_grupos_greedy, k):
    minima_suma = suma_grupos_greedy[0]
    idx_min = 0

    for i in range(1, k):
        suma_act = suma_grupos_greedy[i]

        if suma_act < minima_suma:
            minima_suma = suma_act
            idx_min = i

    return idx_min

def aproximador_pakku(maestros_agua, k):
    if k < 0 or not maestros_agua:
        return None, 0

    grupos = [[] for i in range(k)]
    adic_pakku = 0
    suma_grupos_pakku = [0] * k

    maestros_agua = sorted(maestros_agua, key=lambda x:x[1], reverse=True)

    for maestro in maestros_agua:
        nombre, poder = maestro

        # Grupo con la menor suma actual
        idx_g_min = grupo_min(suma_grupos_pakku, k)

        suma_sin = suma_grupos_pakku[idx_g_min]
        suma_con = suma_sin + poder
```

```
32
33          grupos[idx_g_min].append(maestro)
34          suma_grupos_pakku[idx_g_min] = suma_con
35
36          adic_pakku = adic_pakku - (suma_sin ** 2) + (suma_con ** 2)
37
38      return grupos, adic_pakku
```

5.2. Complejidad

A continuación se detallan las operaciones dentro del algoritmo que aportan complejidad relevante

Complejidad del aproximador:

1. Lo primero que se realiza es la inicialización de una lista de k listas vacías, aportando una complejidad $O(k)$.
2. Posteriormente, se crea una lista con k ceros, representando la suma de los grupos y aportando $O(k)$ en complejidad.
3. A continuación, se ordenan los maestros de forma descendente según su poder, desembocando en una complejidad $O(n\ log(n))$.
4. Se recorre cada maestro y se lo adiciona al grupo con la menor suma actual, por lo que se recorrerán n maestros y los k grupos, dentro de cada iteración también se busca el grupo con la mínima suma dentro de la lista inicializada en el punto 2; lo que aporta una complejidad de $O(k)$ en cada iteración. Al final del recorrido tendremos una complejidad de $O(n * k)$

Por lo tanto, en el peor de los casos donde $n = k$, la complejidad total del algoritmo seria $O(n * k) = O(n^2)$

5.3. Calidad de la aproximación

Para determinar qué tan buena es la solución $A(I)$ obtenida por el algoritmo de aproximación respecto a la solución óptima $z(I)$, definimos el ratio de aproximación como:

$$r(I) = \frac{A(I)}{z(I)}$$

Siempre se cumple que $A(I) \geq z(I)$, por lo que buscamos que $r(I)$ lo mas cercana a 1 posible.

5.3.1. Comparación en instancias manejables

Cuando el tamaño de n y la cantidad de grupos k permiten la ejecución del algoritmo óptimo (Backtracking) en un tiempo razonable, calculamos el ratio de aproximación exacto.

Se utilizaron los datasets provistos obteniendo los siguientes resultados:

Dataset	**Solución óptima (z)**	**Solución Pakku (A)**	**Ratio (A/z)**
8_3.txt	*4298131*	*4298131*	*1*
14_3.txt	*15659106*	*15664276*	*1,0003301*
15_6.txt	*6377225*	*6377501*	*1,0000432*
18_6.txt	*10322822*	*10325588*	*1,0002679*
20_8.txt	*11417428*	*11423826*	*1,0005603*

Cuadro 1: Comparación de soluciones y ratio de aproximación en datasets pequeños

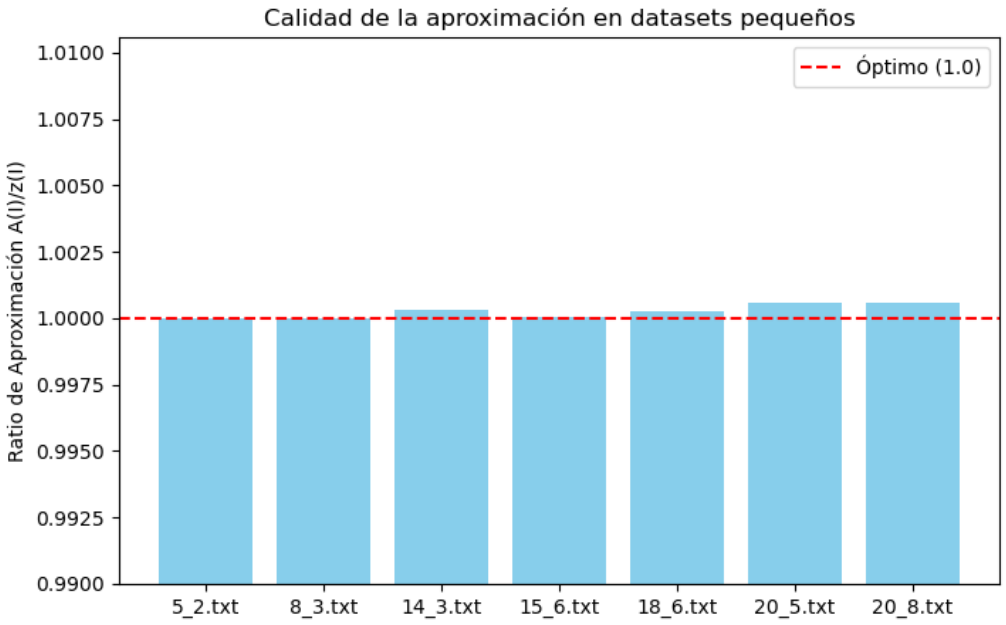

Figura 10: Resultados de $r(I)$ para datasets de la cátedra

Como se observa, para datasets pequeños, el algoritmo de aproximación obtiene resultados extremadamente cercanos al óptimo, con un error prácticamente despreciable.

5.3.2. Comparación en instancias inmanejables

Para las instancias donde el tamaño n (cantidad de maestros) y k (cantidad de grupos) crecen, el algoritmo de Backtracking se vuelve imposible debido a su complejidad exponencial. Por lo tanto, no podemos calcular la solución óptima $z(I)$ para comparar.

Para solucionar esto, utilizamos la estrategia de crear datasets artificiales cuya solución óptima $z(I)$ sea conocida.

Se implementó la funcion `generar_dataset_perfecto` en el archivo `evaluador_aprox.py`, es la encargada de generar los maestros aleatorios y además construir la solución perfecta conocida:

1. La función recibe una cantidad de grupos k y una suma objetivo por grupo `sum_por_grupo`.
2. Por cada uno de los k grupos, genera maestros aleatoriamente.
3. Para el último maestro de cada grupo, en lugar de un valor aleatorio, le asigna el poder exacto restante para que la suma de ese grupo sea exactamente `sum_por_grupo`.
4. Al final, tenemos k grupos que suman exactamente lo mismo, lo cual representa la partición más balanceada posible.

Una vez construido el dataset perfecto, todos los maestros generados se mezclan en una única lista `random.shuffle(maestros_global)`. Esta lista oculta la solución perfecta y se pasa como entrada al `aproximador_pakku`.

Finalmente, se calcula el ratio $r(I)$ comparando el resultado de Pakku $A(I)$ contra el óptimo conocido $z(I)$.

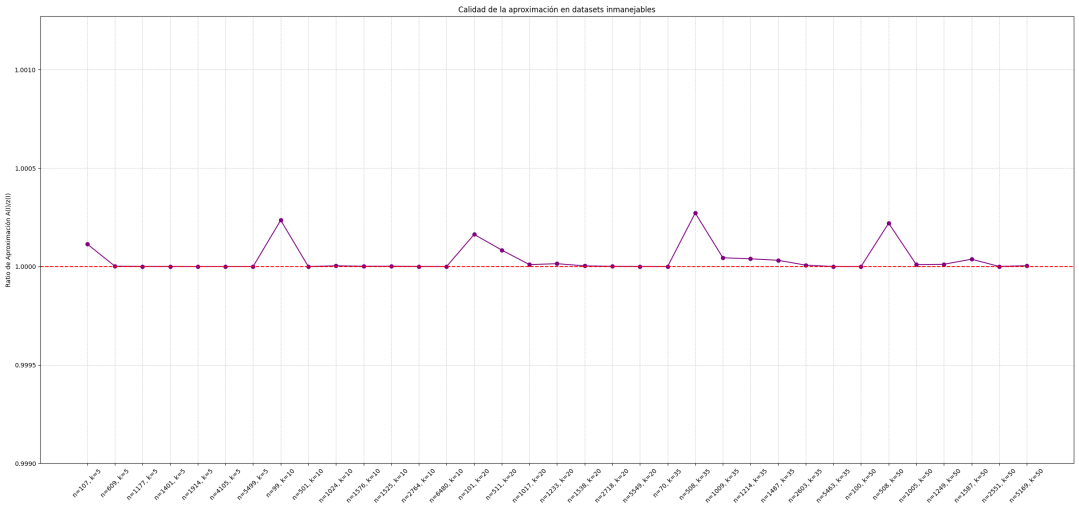

Figura 11: Resultados de $r(I)$ para datasets muy grandes

Como se observa en el gráfico, el algoritmo de Pakku no logra llegar a la solución óptima ($r(i) > 1$). Sin embargo, el error se mantiene notablemente bajo, incluso para volúmenes de datos muy grandes. Si observamos los valores de k y n, podemos llegar a la conclusión que el error tiende a ser ligeramente mayor a medida que k aumenta, pero prácticamente no se ve afectado por n.

5.4. Mediciones de tiempo y comparación con backtracking

5.4.1. Ajuste a la función cuadrática

Para corroborar la complejidad calculada en el punto anterior, se realizaron ciertas mediciones de los tiempos de ejecucion de ese algoritmo para analizar su comportamiento. Luego se estudió su ajuste contra la función $O(n^2)$ y la función $O(n\ log(n))$.

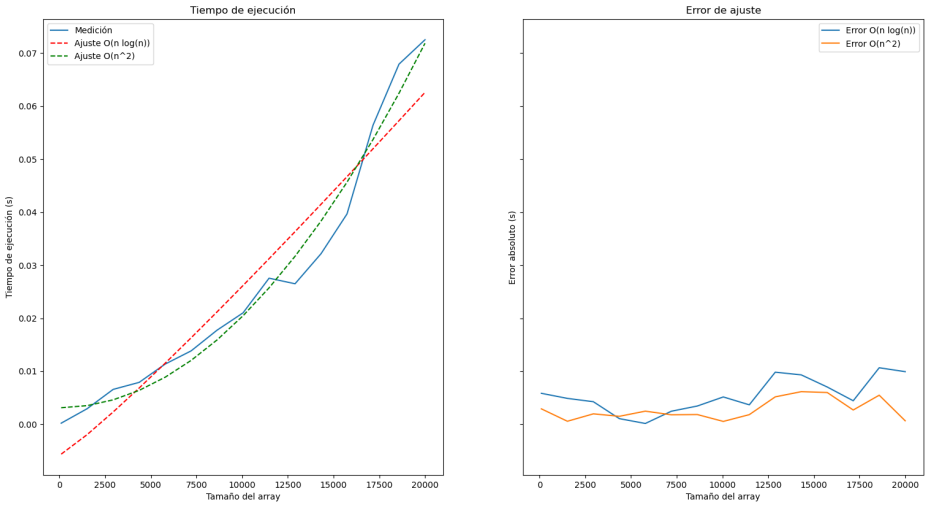

Figura 12: Resultados con *poder* $\in [1, 500]$ y $k = 50$

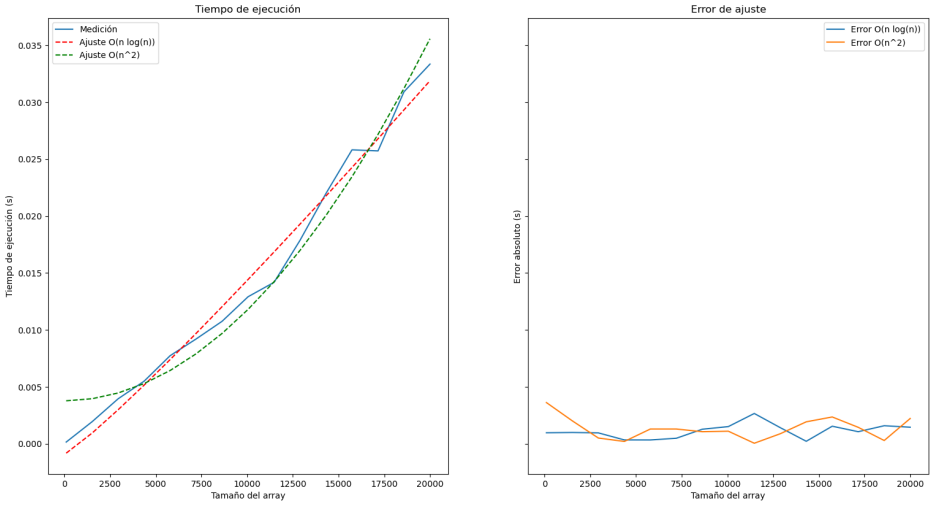

Figura 13: Resultados con *poder* $\in [1, 1000]$ y $k = 30$

Como se puede observar, la función aproximadora de Pakku se ajusta mucho mejor a $O(n^2)$ en lugar de $O(n\ log(n))$.

5.4.2. Comparación contra backtracking

Para estudiar la diferencia de los tiempos de ejecución entre este algoritmo y el implementado con backtracking, se realizaron gráficos de barras donde se modela los diferentes tiempos que tardó cada algoritmo en algunos de los datasets relevantes provistos por la cátedra

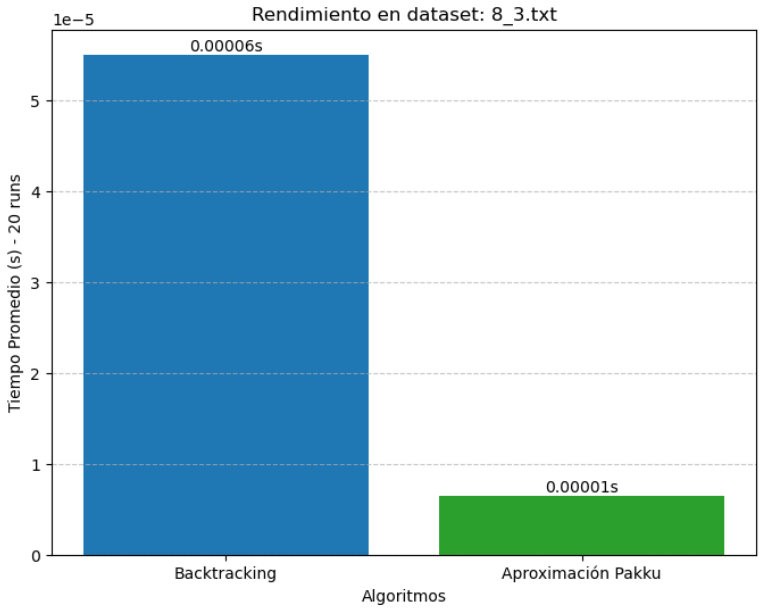

Figura 14: Comparación Pakku vs Backtracking en 8_3.*txt*

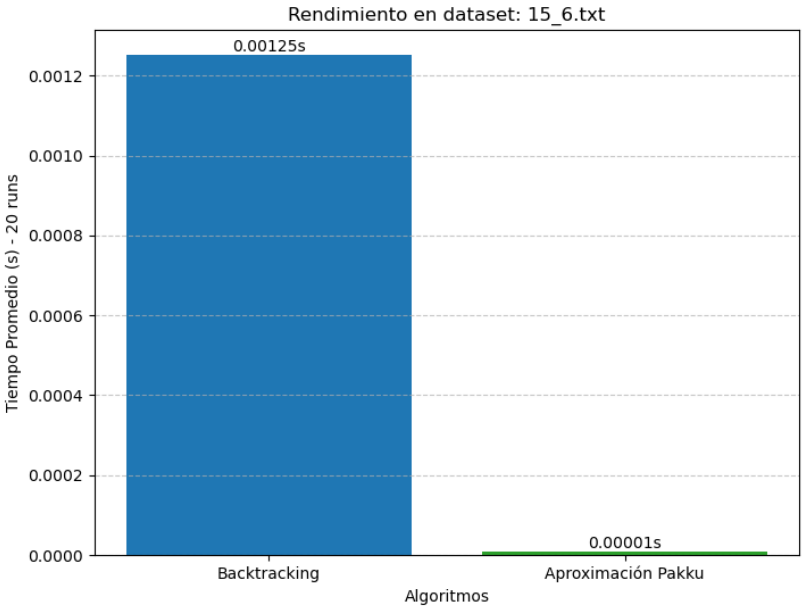

Figura 15: Comparación Pakku vs Backtracking en 15_6.*txt*

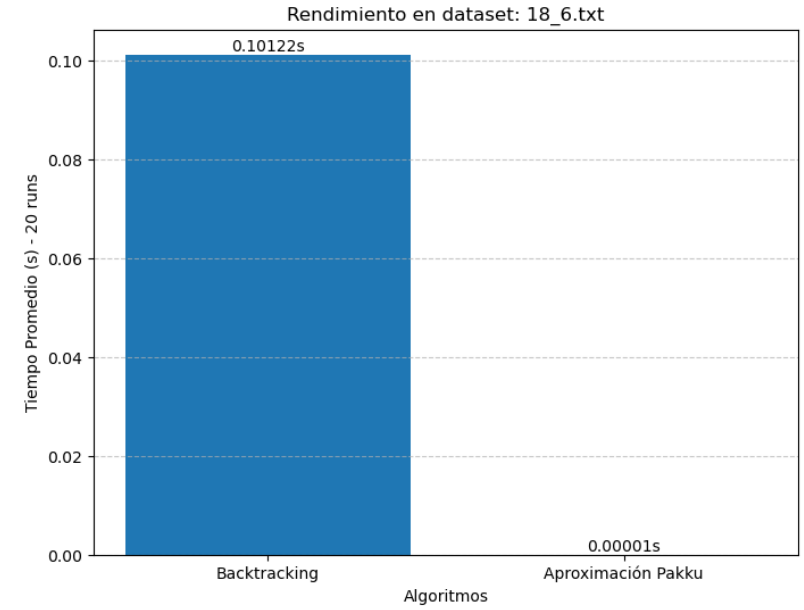

Figura 16: Comparación Pakku vs Backtracking en 18_6.*txt*

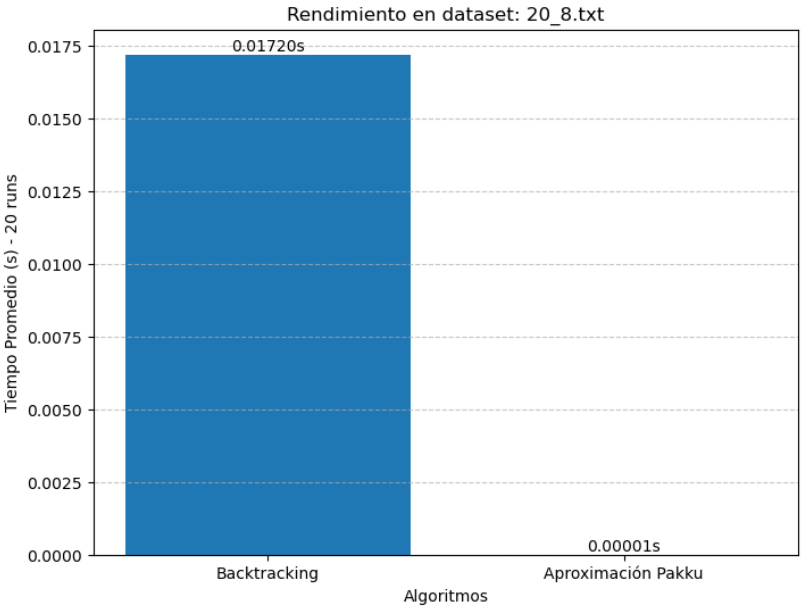

Figura 17: Comparación Pakku vs Backtracking en 20_8.*txt*

En las imágenes anteriores se puede observar la gran diferencia que existe entre los tiempos que tarda cada uno, en especial en los datasets donde los valores k o n (la cantidad de maestros) crece.

6. Aproximador randomizado

Para la implementación de otro aproximador además del aproximador de Pakku descrito en el punto anterior, optamos por hacer un aproximador randomizado basado en la técnica de 'Simulated Annealing', que se inspira en la técnica física del recocido como tratamiento térmico en metales. Elegimos este aproximador por el interés que nos generó su funcionamiento y su naturaleza aleatoria.

6.1. ¿Cómo funciona?

El aproximador cuenta con tres constantes principales sobre las que basa su funcionamiento:

- **Temperatura inicial**
- **Temperatura final**
- **Tasa de enfriamiento**

Este algoritmo va a partir desde la aproximación que calcula el aproximador de Pakku, desde esta aproximación va a hacer cambios aleatorios en sobre las asignaciones de los maestros en cada grupo. Elige un grupo aleatorio que no este vacío, dentro de ese grupo elige un maestro aleatorio y luego lo envía a otro grupo también elegido al azar.

Al hacer los cambios pueden darse dos situaciones diferentes:

Caso 1: el cambio aleatorio mejoró la solución aproximada que había calculado el algoritmo de Pakku, asi que se toma la decisión de guardar ese cambio.

Caso 2: el haber cambiado al maestro seleccionado de grupo causó que empeore la solución, sin embargo puede darse la situación donde una mala decisión a corto plazo pueda resultar en alcanzar una mejor solución a medida que se siguen realizando cambios. Por esto, simulated annealing va a decidir si tomar los cambios que empeoran la solución con la esperanza de que al hacerlos pueda alcanzar una mejor al largo plazo.

La decisión ligada a una probabilidad, que depende tanto de la temperatura actual como de que tan malo es el cambio. Cuando la temperatura es alta, el algoritmo mantiene probabilidades altas de tomar los cambios malos sin tener tanto en cuenta la calidad del mismo, pero a medida que baja la temperatura, la probabilidad de tomar cambios que provocaron un empeoramiento significativo en la solución decrece significativamente.

El código sobre el que esta implementado el algoritmo es el siguiente:

```
def simulated_annealing(maestros_agua, k):
    mejor_sol, adic_mejor = aproximador_pakku(maestros_agua, k)

    sol_actual = [list(g) for g in mejor_sol]
    adic_actual = adic_mejor
    suma_grupos_actual = calcular_suma_grupos(sol_actual, k)

    t_actual = T_INICIAL

    while t_actual >= T_FINAL:
        g1 = elegir_random_grupo_no_vacio(sol_actual)

        idx = random.randrange(len(sol_actual[g1]))
        maestro = sol_actual[g1][idx]
        nombre, poder = maestro

        g2 = random.randrange(k)
        while g1 == g2:
            g2 = random.randrange(k)

```

```
        suma_g1_vieja = suma_grupos_actual[g1]
        suma_g2_vieja = suma_grupos_actual[g2]

        suma_g1_nueva = suma_g1_vieja - poder
        suma_g2_nueva = suma_g2_vieja + poder

        adic_nuevo = adic_actual - (suma_g1_vieja**2 + suma_g2_vieja**2) + (
    suma_g1_nueva**2 + suma_g2_nueva**2)
        delta_adic = adic_nuevo - adic_actual

        if acepta_cambio(delta_adic, t_actual):
            adic_actual = adic_nuevo
            suma_grupos_actual[g1] = suma_g1_nueva
            suma_grupos_actual[g2] = suma_g2_nueva

            sol_actual[g1].pop(idx)
            sol_actual[g2].append(maestro)

            if adic_actual < adic_mejor:
                adic_mejor = adic_actual
                mejor_sol = [list(g) for g in sol_actual]

        t_actual *= TASA_ENFRIAMIENTO

    return mejor_sol, adic_mejor
```

Como se puede observar, al final de cada iteración del bucle `while`, la temperatura decrece multiplicándola por una `TASA_ENFRIAMIENTO` (en nuestro algoritmo usamos 0.995, un valor positivo cercano a 1).

La toma de decisiones está encapsulada en la función `acepta_cambio`:

```
def acepta_cambio(delta_adic, t_actual):
    if delta_adic < 0:
        return True
    else:
        p = math.exp(-delta_adic / t_actual)
        if random.random() < p:
            return True
    return False
```

Si el delta es un valor negativo, el cambio aleatorio terminó mejorando la solución, por lo que la acepta siempre. Lo interesante se ve cuando el cambio no reflejó un cambio positivo en el coeficiente, en cuyo caso calcula la probabilidad de aceptarlo usando la distribución probabilística de Boltzmann con los parámetros `delta_adic` y `t_actual`, donde se puede ver su comportamiento:

1. Temperaturas altas permiten cambios no tan buenos
2. Las temperaturas bajas disminuyen mucho la probabilidad para los cambios que empeoraron mucho la solución

6.2. Comparación contra el aproximador de Pakku

Como este aproximador ya parte de la solución que encontró el aproximador de Pakku, y a partir de esta busca mejorarla, es imposible que el resultado de la aproximación sea peor que el calculado por el algoritmo de Pakku. La única posibilidad es que, dependiendo de los números aleatorios generados, sea la misma aproximación o que sea mejor. A continuación se muestra un gráfico donde se comparan los resultados de la cota de aproximación para Pakku (violeta) y Simulated Annealing (Verde), donde efectivamente se observa que es igual o mejor.

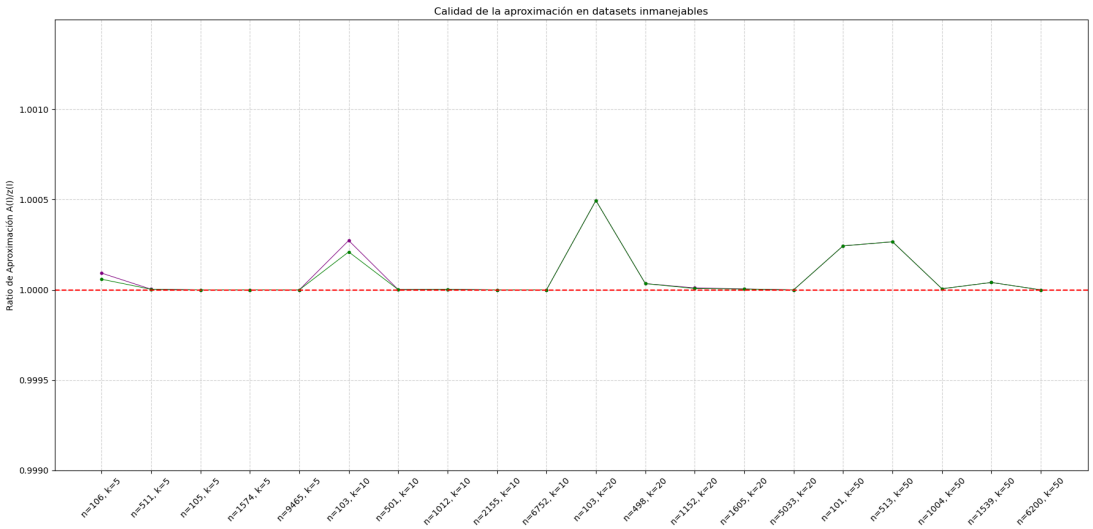

Figura 18: Resultados de $r(I)$ para datasets muy grandes

En el gráfico anterior se puede dimensionar que si bien en algunas ejecuciones logra mejorar la solución proporcionada por el aproximador de Pakku, no lo hace en un porcentaje muy grande. Por lo que a modo de prueba, también medimos la diferencia de los algoritmos pero con un cambio, ahora el aproximador Greedy de Pakku no ordena a los maestros por poder decreciente y Simulated Annealing también comenzará a hacer cambios aleatorios desde el resultado del algoritmo de Pakku alterado.

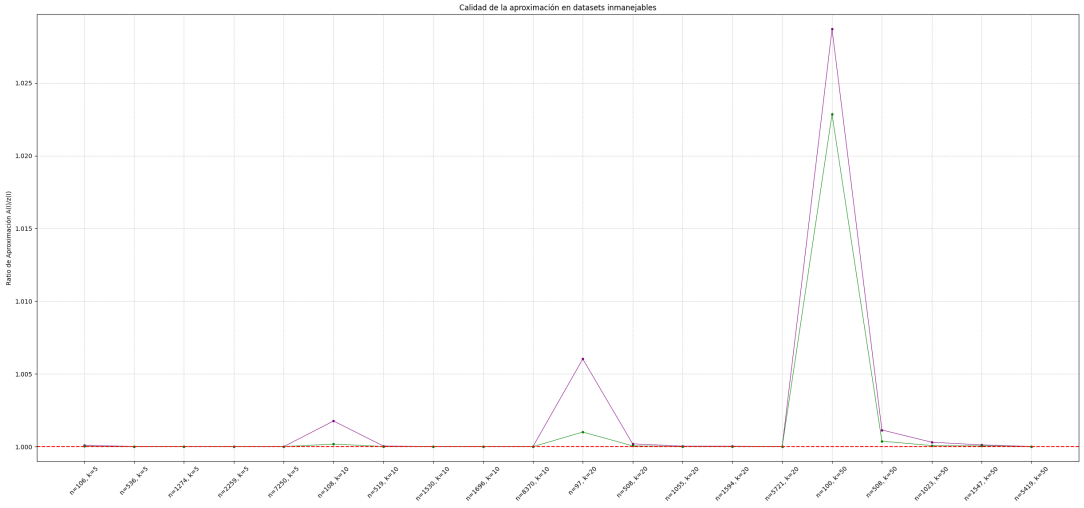

Figura 19: Resultados de $r(I)$ para datasets muy grandes con Pakku modificado

En ese gráfico si se puede notar que como el aproximador de Pakku alterado no alcanza una solución de buena calidad, es mucho mas fácil para simulated annealing lograr mejorarla en un porcentaje mas notable.

6.3. Complejidad del algoritmo

La complejidad de la implementación de simulated annealing mostrada en este informe está dada por sus principales instrucciones:

1. Obtener la solución del aproximador de Pakku. Como se mostró en la sección 5 del informe, esto puede aportar una complejidad de $O(n^2)$ en el peor de los casos.
2. Hacer una copia del arreglo de asignaciones aporta una complejidad $O(n + k)$
3. Calcular la suma de los grupos en la asignación devuelta por el aproximador de Pakku aporta $O(n + k)$
4. Lo siguiente es el análisis de la complejidad del bucle `while`, cuya cantidad de iteraciones es una constante definida por valores constantes en el algoritmo. La cantidad de iteraciones totales, definida como I, de este bucle se calcula de la siguiente manera:

   Para simplificar las siguientes ecuaciones se define α como la tasa de enfriamiento

$$TINICIAL \cdot (\alpha)^I \approx TFINAL \tag{7}$$

   Por lo tanto, despejando I, se obtiene la siguiente cantidad de iteraciones:

$$I \approx \frac{\log(\frac{T_{FINAL}}{T_{INICIAL}})}{\log(\alpha)} \tag{8}$$

   Es decir, la cantidad de iteraciones del bucle no depende de los valores de entrada del programa k o n, por lo tanto puede considerarse como que es una cantidad de iteraciones constante.

   Dentro del bucle se realizan varias operaciones de complejidad $O(n)$ o $O(n + k)$.

Por lo tanto, la complejidad del algoritmo de simulated annealing esta marcada principalmente por el llamado al aproximador de Pakku, por lo que su complejidad teórica es $O(n^2)$.

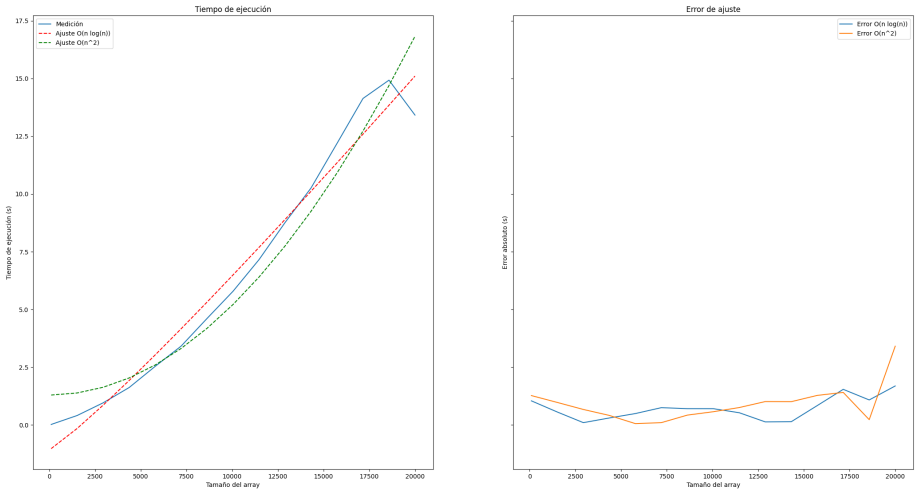

Figura 20: Mediciones y ajuste de simulated annealing