Teoría de algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico 3

Problemas NP-Completos para la defensa de la Tribu del Agua

---

6 de mayo de 2024

Integrantes:

- Matías Vázquez Morales (111083)
- Nestor Fabian Palavecino Arnold (108244)

Introducción

En este informe, presentaremos diferentes algoritmos para la resolución de la ecuación

$$min \sum_{i=1}^{k} (\sum_{x_j} x_j)^2$$

Presentada en el trabajo practico, para esto se utilizó técnicas de backtracking, greedy y programación lineal,

Adicionalmente, analizaremos:

- La complejidad temporal y espacial de la solución
- Variabilidad de algunos valores en el algoritmo planteado
- Tiempos de ejecución para corroborar la complejidad teórica indicada

Los algoritmos planteados se encuentran en el repositorio de GitHub:

https://github.com/NestorPala/tda-tp3

El Problema de la Tribu del Agua

Vamos a resolver a continuación el Problema de la Tribu del Agua, en su versión "problema de decisión". El problema recibe 3 variables:

- Un conjunto de valores X_j que representan los valores de habilidad de los guerreros
- Un valor k que determina la cantidad de subgrupos a armar de los guerreros
- Un valor B que representa una cota para el coeficiente de habilidad

Necesitamos demostrar a continuación 2 postulados:

- El P.T.A se encuentra en NP
- El P.T.A es un problema NP-Completo

El P.T.A se encuentra en NP

Para demostrar que P.T.A. está en NP, necesitamos encontrar un validador del problema de decisión que funcione en tiempo polinomial.

El **validador** que nosotros proponemos es:

- Agarrar los valores X_j y B y "meterlos a la licuadora" (S_j representa el subgrupo j):

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

- Si se cumple la ecuación de arriba, devolver `true`, caso contrario `false.`

Complejidad del validador: **O(n)**, con n la cantidad de valores X_j

El código del validador es el siguiente:

```python
from functools import reduce


def validador_tribu_agua(grupos: list[list[int]], b: int):
    coeficiente = sum(list(map(lambda x: sum(x)**2, grupos)))
    return coeficiente <= b
```

El P.T.A es un problema NP-Completo

Luego, para demostrar que el problema es NP-Completo, necesitamos poder reducir un problema NP-Completo a resolver el PTA (TribuAgua).

La **reducción** que nosotros proponemos es la siguiente:

$$SubsetSum \leq_p TribuAgua$$

Para ello, arrancamos de la inecuación del problema de decisión del Problema de la Tribu del Agua:

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

Y la modificamos tomando $k = n$, con n el tamaño del array de guerreros:

$$\sum_{i=1}^{n} {x_i}^2 \leq B$$

Esta inecuación la vamos a llevar a la forma de Subset Sum:

$$\sum_{i=1}^{p} x_i' = w$$

Aquí, p es el tamaño del subset y w el valor deseado a igualar.

Lo primero que haremos, será obtener los valores X prima:

$$x_i^2 = x_i' \;\rightarrow\; x_i = \sqrt{x_i'}$$

Luego:

$$B = w$$

Y nos queda:

$$\sum_{i=1}^{p} (\sqrt{x_i'})^2 \leq w$$

Ahora, necesitamos convertir ese signo “menor o igual” en un signo “igual”. Para esto construimos una nueva desigualdad:

- Sumando (– p * W) a ambos lados
- Multiplicando por (-1)

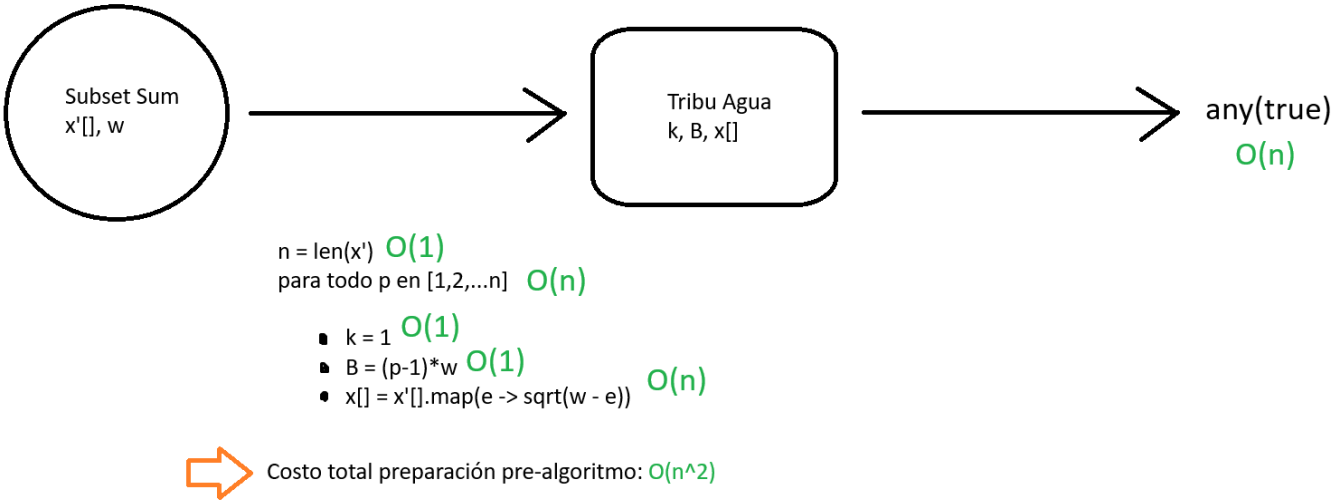
Al hacer esto, nos queda que para que se cumpla la desigualdad, sí o sí la suma de los X prima **tiene que ser igual a W.** Esto es lo que necesitamos para Subset Sum.

Luego, utilizamos la misma estrategia de antes para convertir los X prima en valores utilizables para el *solver* de PTA.

La inecuación final nos queda:

$$\sum_{i=1}^{p} (\sqrt{(w - x_i')})^2 \leq (p-1) \cdot w \qquad \forall p \in [1,2, \dots n]$$

La reducción entonces es la siguiente:



Con esto demostramos que Subset Sum se puede reducir a PTA, por lo tanto, PTA es NP-Completo.

Algoritmo de Programación Lineal Entera

Para resolver el Problema de la Tribu del Agua utilizando programación lineal entera, utilizamos la **opción de aproximación** mediante la resta entre los equipos con más y con menos puntos de habilidad respectivamente.

El **enfoque de la solución** se basa en tener k ecuaciones, cada una representando uno de los subsets de la solución. En cada una de esas ecuaciones, hay n términos, cada uno de ellos compuesto por los puntos de habilidad de un guerrero multiplicado por una variable binaria que determina si el guerrero entra o no en ese subset.

Luego, utilizamos inecuaciones para ordenar los subsets por su suma total, de mayor a menor, y

tomamos el primer y último subset para hacer la resta que queremos minimizar.

Esto lo representamos utilizando las siguientes ecuaciones:

$$1)\ \forall j \in [1,2,\dots k] \quad S_j = \sum_{i=1}^{n} \alpha_{ji} \cdot X_i$$

$$2)\ \forall j \in [2,3,\dots k] \quad S_1 \geq S_k$$

$$3)\ \forall j \in [1,2,\dots k-1] \quad S_k \leq S_j$$

$$4)\ \forall j \in [1,2,\dots n] \quad \sum_{j=1}^{k} \alpha_{ij} = 1$$

$$5)\ \min(S_1 - S_k)$$

Las ecuaciones 1) a 4) son las funciones restricción y la expresión 5) es la función objetivo, la cual debemos minimizar para obtener la solución.
En dichas ecuaciones

- j recorre las ecuaciones de los subsets
- i recorre las variables binarias asociadas a los puntos de habilidad de cada guerrero
- n es la cantidad de guerreros del problema

La ecuación 1) nos asegura que la solución aproximada contenga exactamente k subsets.

La ecuación 2) nos asegura que S_1 sea la suma total del subset más grande.

La ecuación 3) nos asegura que S_k sea la suma total del subset más pequeño.

La ecuación 4) nos asegura que cada guerrero aparezca exactamente 1 vez en cada subset.

La **complejidad** del algoritmo es $O(2^n)$, que es la complejidad estándar para P.L.E.

Las **mediciones** tomadas para este algoritmo utilizando los archivos de prueba de la catedra son los siguientes:

ARCHIVO	SOLUCION OPTIMA	SOLUCION APROXIMADA	RELACION DE APROXIMACION	TIEMPO TOMADO (segundos)
5_2	1894340	1894340	1	0.0182802677154541
6_3	1640690	1640690	1	0.11351966857910156
6_4	807418	807418	1	0.023112058639526367
8_3	4298131	4298131	1	0.11633992195129395

10_3	385249	385249	1	0.26928257942199707
10_5	355882	355890	1.00003	1.3504464626312256
11_5	2906564	2906564	1	1.7504420280456543
14_3	15659106	15659106	1	1.0376636981964111
14_4	15292055	15292055	1	3.261610746383667
14_6	10694510	10694510	1	131.55025434494019
15_4	4311889	4311889	1	3.5936622619628906

En esta tabla, la relación de aproximación fue calculada con la fórmula:

$$\frac{SOLUCION\ APROXIMADA}{SOLUCION\ OPTIMA}$$

Dando una cota superior de 1.00003, siendo esta la mayor relación de aproximación medida.

Algoritmo de Backtracking

El algoritmo de backtracking se encuentra en el siguiente enlace , en este algoritmo nos da siempre la mejor solución al problema del trabajo practico, pero su complejidad temporal no es la mejor.

```
# n = numero de maestros, k = numero de grupos, x= lista de fuerzas de los maestros
# grupo = lista de sumas de fuerzas de los grupos, pos = posicion actual en la lista
de maestros
# minimo = minimo encontrado hasta ahora
def backtracking_(n, k, x, grupo, asignacion, pos, minimo):
    # si se asignaorn todos los maestros a un grupo, calcula la suma de los
cuadrados de las sumas de las fuerzas de los grupos
    if pos == n:
        suma = sum(i**2 for i in grupo)
        if suma < minimo[0]:
            minimo[0] = suma
            minimo[1] = asignacion[:]
    else:
        # bucle que recorre grupos
        for i in range(k):
            # asigna al maestro actual al grupo
            grupo[i] += x[pos]
            asignacion[pos] = i
            # verifica si la suma es manor al minimo actual, si es asi sigue con el
sig maestro
```

```
            if sum(j**2 for j in grupo) < minimo[0]:
                # llama la siguiente posicion
                backtracking_(n, k, x, grupo, asignacion, pos + 1, minimo)
            # deshace la asignacion del maestro
            grupo[i] -= x[pos]
```

esta es la parte más compleja del algoritmo, la complejidad de un algoritmo de backtracking es difícil de calcular.
Sin embargo, en el peor de los casos cada maestro puede ser asignado a cualquiera de los k grupos, donde esto es k^n combinaciones posibles, cada llamada calcula la suma de los cuadrados donde esto tiene un costo de k. por lo tanto la complejidad total es O(k^(n+1))

```
def backtracking(cantidad_de_grupos, nombre_y_habilidad):
    # numero total de maestros
    n = len(nombre_y_habilidad)
    # crea lista con las fuerzas de los maestros
    x = [habilidad for _, habilidad in nombre_y_habilidad]
    grupo = [0]*cantidad_de_grupos
    asignacion = [0]*n
    minimo = [float('inf'), []]
    backtracking_(n, cantidad_de_grupos, x, grupo, asignacion, 0, minimo)
    return minimo[0], minimo[1]
```

esta función prepara los parámetros y llama a la función backtracking_, la complejidad esta dominada por esta llamada, donde la complejidad es O(k^(n+1))

```
def escribir_archivo(coeficiente, asignacion, nombre_y_habilidad):
    res = {}

    try:
        solucion = open(ARCHIVO_ESCRIBIR, "w")
    except:
        print("error al abrir el archivo")
        return

    grupos = [[] for _ in range(max(asignacion) + 1)]
    for (nombre, _), grupo in zip(nombre_y_habilidad, asignacion):
        grupos[grupo].append(nombre)

    for i, grupo in enumerate(grupos, 1):
        key_ = f'Grupo {i}'
        solucion.write(f'{key_}: {", ".join(grupo)}\n')
        res[key_] = grupo

    solucion.write(f'Coeficiente: {coeficiente}')
    solucion.close()
```

```
    return res
```

esta función tiene complejidad O(n), donde n es el número de maestros.

```
def tribu_agua_backtracking(guerreros, k):
    start_time = time.time()
    coeficiente, asignacion = backtracking(k, guerreros)
    end_time = time.time()

    tiempo_total = end_time - start_time

    res = escribir_archivo(coeficiente, asignacion, guerreros)
    print("la solucion se encuentra en el archivo solucion.txt")
    print(f"la funcion de backtraking tardo {tiempo_total} segundos.")

    res = get_sorted_guerreros(res, guerreros)

    return res, coeficiente
```

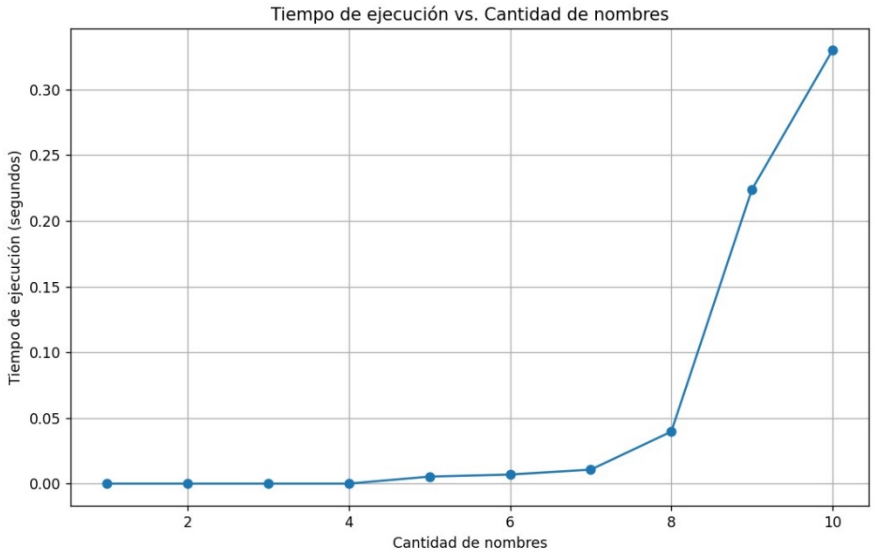
esta función hace el llamado a las funciones anteriores, por lo que su complejidad temporal es la suma de las funciones, es decir O(n + k^(n+1))

```
def get_sorted_guerreros(res: dict[str, list[str]], guerreros: list[str, int]):
    guerreros_ = {}
    for tuple_ in guerreros:
        key_ = tuple_[0]
        value = tuple_[1]
        guerreros_[key_] = value

    for key, value in res.items():
        res[key] = sorted(value, key=lambda name: guerreros_[name], reverse=True)

    return res
```

esta función se puede dividir en 2 partes, una es la construcción del diccionario guerreros_ que tiene una complejidad O(n) y la otra es el ordenamiento de listas en res que tiene una complejidad O(m * k log k) donde m es la cantidad de claves en res y k es la cantidad de elementos de cada lista. Por lo tanto la complejidad total de esta función en O(n + m*k log k) donde n es el número de elementos en la lista guerreros, m es el número de claves en res y k es el tamaño de las listas a cada clave en res

Por lo tanto, la complejidad del algoritmo es O(k^(n+1)) donde n es el número de maestros y k es el número de grupos. Sin embargo, como es un algoritmo de backtracking, la complejidad podría varias dependiendo de los datos de entrada y de las podas.

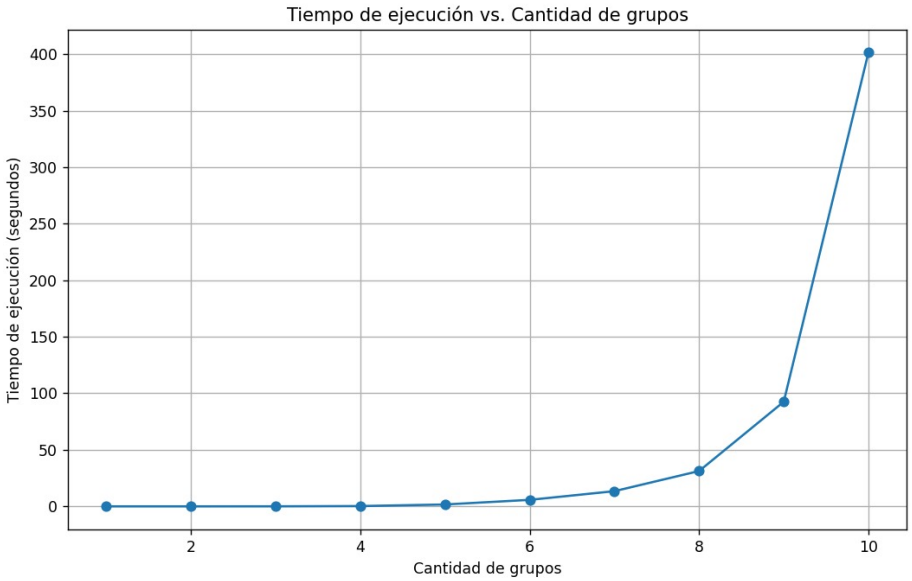
Gráfico del algoritmo en función de la cantidad de nombres:



En este grafico para todas las ejecuciones se tiene en cuenta que la cantidad de grupos para todas las ejecuciones son 4.

Gráfico del algoritmo en función de la cantidad de grupos:

En este grafico para todas las ejecuciones se tiene en cuenta que la cantidad de personas para todas las ejecuciones es 10



Algoritmo de aproximación del Maestro Pakku

El algoritmo de aproximación dado en el punto 5 se encuentra en el siguiente enlace.

```
def aproximacion_Maestro_Pakku(guerreros, k):
    # Ordenamos de mayor a menor los maestros en función de su habilidad o fortaleza
    habilidad_ordenada = sorted(guerreros, key=lambda x: int(x[1]), reverse=True)
    return aproximacion(k, habilidad_ordenada)
```

esta función tiene una complejidad temporal de O(n log n), donde n es el número total de guerreros. Pero al hacer el llamado a la función aproximación también depende de la complejidad de esta función.

```
def tribu_agua_aproximacion(guerreros, k):
    start_time = time.time()
    grupos, coeficiente = aproximacion_Maestro_Pakku(guerreros, k)
    end_time = time.time()

    total_time = end_time - start_time

    escribir_archivo(grupos, coeficiente, ARCHIVO_ESCRIBIR)
    print("la solucion se encuentra en el archivo solucion_aproximada.txt")
    print(f"el tiempo que tarda el algortimo es {total_time}")

    res = {}
    i = 0
    for grupo in grupos:
        i+=1
        key = f"Grupo {i}"
        res[key] = []
        for e in grupo:
            res[key].append(e)

    res = get_sorted_guerreros(res, guerreros)

    return res, coeficiente
```

esta función tiene una complejidad O(n log n) por el ordenamiento y la escritura en el archivo, hace el llamado a la función aproxiamacion_Maestro_Pakku que también tiene una complejidad O(n log n), pero esta hace el llamado a la función aproximación asique la complejidad depende de esta función.

```python
def aproximacion(cantidad_de_grupos, habilidad_ordenada):
    grupos = [[] for _ in range(cantidad_de_grupos)]
    sumas_cuadradas = [0] * cantidad_de_grupos

    for nombre, habilidad in habilidad_ordenada:
        grupo_minimo = min(range(cantidad_de_grupos), key = lambda i:
sumas_cuadradas[i])

        grupos[grupo_minimo].append(nombre)

        sumas_cuadradas[grupo_minimo] += habilidad

    minimo = sum(j**2 for j in sumas_cuadradas)

    return grupos, minimo
```

esta función tiene una complejidad O(nk) donde n es el número de maestros y k es el número de grupos. Para cada maestro, busca el grupo con la suma mínima de habilidades, que toma tiempo O(k) y hace esto para todos los n maestros

```python
def get_sorted_guerreros(res: dict[str, list[str]], guerreros: list[str, int]):
    guerreros_ = {}
    for tuple_ in guerreros:
        key_ = tuple_[0]
        value = tuple_[1]
        guerreros_[key_] = value

    for key, value in res.items():
        res[key] = sorted(value, key=lambda name: guerreros_[name], reverse=True)

    return res
```

esta función se puede dividir en 2 partes, una es la construcción del diccionario guerreros_ que tiene una complejidad O(n) y la otra es el ordenamiento de listas en res que tiene una complejidad O(m * k log k) donde m es la cantidad de claves en res y k es la cantidad de elementos de cada lista. Por lo tanto la complejidad total de esta funcnion en O(n + m*k log k) donde n es el numero de elementos en la lista guerreros, m es el numero de claves en res y k es el tamaño de las listas a cada clave en res

En conclusión la complejidad del algoritmo de aproximación es O(n log n + nk + m * k log k), esto es una mejora significativa en complejidad comparado con el algoritmo de backtracking pero a diferencia, este no da la solución óptima, sino que da una aproximación a esta.

ARCHIVO	SOLUCION OPTIMA	SOLUCION APROXIMADA	RELACION DE APROXIMACION
5_2	1894340	1894340	1
6_3	1640690	1640690	1
10_5	355882	355882	1
10_10	172295	172295	1
14_6	10694510	10700172	1,00053
17_5	15974095	15975947	1,00012
17_10	5427764	5430512	1,00051
18_8	11971097	12000279	1,00244
20_8	11417428	11423826	1,00056

En esta tabla se miden los resultados óptimos y aproximados de 9 archivos dados por la catedra, la relación de aproximación fue calculada con

$$\frac{SOLUCION\ APROXIMADA}{SOLUCION\ OPTIMA}$$

Dando una cota superior de 1.00244, siendo esta la mayor relación de aproximación medida

Algoritmo de aproximación greedy

El algoritmo de aproximación greedy se encuentra en el siguiente enlace, este algoritmo es similar al algoritmo dado en el punto 5, con la diferencia que no ordena a los maestros por habilidad, pero reparte a los maestros de igual manera.

```python
def get_sorted_guerreros(res: dict[str, list[str]], guerreros: list[str, int]):
    guerreros_ = {}
    for tuple_ in guerreros:
        key_ = tuple_[0]
        value = tuple_[1]
        guerreros_[key_] = value

    for key, value in res.items():
        res[key] = sorted(value, key=lambda name: guerreros_[name], reverse=True)

    return res
```

esta función se puede dividir en 2 partes, una es la construcción del diccionario guerreros_ que tiene una complejidad O(n) y la otra es el ordenamiento de listas en res que tiene una complejidad O(m *

k log k) donde m es la cantidad de claves en res y k es la cantidad de elementos de cada lista. Por lo tanto la complejidad total de esta funcnion en O(n + m*k log k) donde n es el numero de elementos en la lista guerreros, m es el numero de claves en res y k es el tamaño de las listas a cada clave en res

```
def aproximacion(cantidad_de_grupos, habilidad_ordenada):
    grupos = [[] for _ in range(cantidad_de_grupos)]
    sumas_cuadradas = [0] * cantidad_de_grupos

    for nombre, habilidad in habilidad_ordenada:
        grupo_minimo = min(range(cantidad_de_grupos), key = lambda i:
sumas_cuadradas[i])

        grupos[grupo_minimo].append(nombre)

        sumas_cuadradas[grupo_minimo] += habilidad

    minimo = sum(j**2 for j in sumas_cuadradas)

    return grupos, minimo
```

esta función tiene una complejidad O(nk) donde n es el número de maestros y k es el número de grupos. Para cada maestro, busca el grupo con la suma mínima de habilidades, que toma tiempo O(k) y hace esto para todos los n maestros

```
def tribu_agua_greedy(guerreros, k):
    start_time = time.time()

    # En este caso no ordenamos los guerreros por nivel de habilidad
    grupos, coeficiente = aproximacion(k, guerreros)

    end_time = time.time()
    total_time = end_time - start_time

    escribir_archivo(grupos, coeficiente, ARCHIVO_ESCRIBIR)

    print("la solucion se encuentra en el archivo solucion_aproximada.txt")
    print(f"el tiempo que tarda el algortimo es {total_time}")

    res = {}
    i = 0
    for grupo in grupos:
        i+=1
        key = f"Grupo {i}"
        res[key] = []
        for e in grupo:
            res[key].append(e)
```

```
    res = get_sorted_guerreros(res, guerreros)

    return res, coeficiente
```

esta función hace el llamado a todas las funciones anteriores, por lo que su complejidad temporal es la suma de las anteriores O(n + nk)

En conclusión la complejidad del algoritmo greedy de aproximación es O(n + nk) al no tener que ordenar los maestros por habilidad, puede ser más eficiente en términos de complejidad temporal, pero el algoritmo de aproximación del punto 5 puede dar soluciones con mejor calidad

ARCHIVO	SOLUCION OPTIMA	SOLUCION APROXIMADA	RELACION DE APROXIMACION
5_2	1894340	1918996	1,013015614937128
6_3	1640690	1696190	1,033827231225887
10_5	355882	383346	1,077171646781798
10_10	172295	172295	1
14_6	10694510	10924812	1,021534600463228
17_5	15974095	16139807	1,010373795823801
17_10	5427764	5936050	1,093645560123837
18_8	11971097	12575883	1,05052051620666
20_8	11417428	12223294	1,070582096072776

En esta tabla se miden los resultados óptimos y aproximados de 9 archivos dados por la catedra, la relación de aproximación fue calculada con

$$\frac{SOLUCION\ APROXIMADA}{SOLUCION\ OPTIMA}$$

Dando una cota superior de 1,093645560123837, siendo esta la mayor relación de aproximación medida
Comparando con la aproximación del punto anterior, nos da una diferencia de 0,091205560123837 en las cotas, dando un mejor resultado en el algoritmo del punto 5 que este.

Conclusiones

Como se pudo observar en el presente trabajo práctico:

- El algoritmo de backtracking claramente tiene una relación exponencial en el tiempo de ejecución vs la entrada del usuario.

- Los algoritmos de aproximación por programación lineal entera y el “algoritmo del maestro Pakku” son muy buenas aproximaciones, sobre todo la de PL, que casi siempre da el resultado correcto.

- La aproximación greedy propuesta no es muy buena pero el error siempre es de menos del 10%. Eso sí, la velocidad de ejecución denota claramente su naturaleza cuadrática, porque es extremadamente veloz.

- Los tiempos de ejecución aumentan significativamente al aumentar el número de grupos a usar para dividir a los guerreros.