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

Facultad de Ingeniería de la Universidad de Buenos Aires

Teoría de Algoritmos

Cátedra Buchwald-Genender

Gómez Belis, Sofía
Padrón: 109358
email: sgomezb@fi.uba.ar

Llanos Pontaut, Valentina
Padrón: 104413
email: vllanos@fi.uba.ar

Orsi, Tomas Fabrizio
Padrón: 109735
email: torsi@fi.uba.ar

Indice

1 Introducción

1.1 Descripción y objetivo

Continuando con el ataque de la Nación del Fuego sobre el resto de las naciones, esta vez es la Tribu del Agua la que requiere de nuestra ayuda para defenderse.

Cada maestro agua tiene una fuerza o habilidad positiva x_i, y contamos con el conjunto de todos los valores $(x_i, x_2, \ldots, x_n)$. Basándonos en estos, el maestro Pakku desea separar los maestros en `k` grupos $(S_1, S_2, \ldots, S_k)$ parejos tal que cuando un grupo se canse, entrará el siguiente en el combate, obteniendo un ataque constante que les permita salir victoriosos, aprovechando también la ventaja del agua por sobre el fuego.

Para que los grupos estén lo más parejos posibles, nos han encomendado minimizar la adición de los cuadrados de las sumas de las fuerzas de los grupos:

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

En este trabajo desarrollaremos algoritmos de backtracking, programación lineal y posibles aproximaciones buscando resolver el problema de optimización planteado con el objetivo de ayudar a los mestros de la Tribu del Agua a derrotar a la Nación del Fuego. También nos dedicaremos a demostrar que el problema de la tribu del agua es NP-Completo.

2 Demostración de problema NP-Completo

2.1 Problema NP

2.2 Reducción

3 Algoritmos y análisis de complejidad

El problema de optimización de la tribu del agua fue resuelto utilizando distintas técnicas de programación. En las próximas secciones presentaremos el código correspondiente y analizaremos la complejidad temporal de cada uno de los algoritmos planteados:

- Backtracking
- Programación Lineal
- Aproximación propuesta por la cátedra
- Aproximación adicional

3.1 Complejidad lectura de archivos

A continuación mostramos la función principal de lectura de archivos.

```
9  FLAGPL = "--pl"
10 FLAGAPROXCATEDRA = "--a1"
11 FLAGAPROXADICIONAL = "--a2"
12
13
14 algoritmos = {
15     FLAGBACKTRACKING : problema_tribu_del_agua_bt,
16     FLAGBTGREEDY : problema_tribu_del_agua_bt_greedy,
17     FLAGPL : problema_tribu_del_agua_pl,
18     FLAGAPROXADICIONAL: problema_tribu_del_agua_aprox_adicional
19 }
20
21 def generarTestDe(archivo):
22     k = None
23     maestros_y_habilidades = []
24
```

La función lee la línea que contiene la cantidad de conjuntos de maestros a crear. Una vez hecho eso, lee n líneas para almacenar los distintos valores x_i de tuplas (nombre, habilidad). Esto tiene una complejidad temporal $O(n)$.

3.2 Algoritmo de Backtracking

Debido al alto tiempo de ejecución de nuestro algoritmo inicial de backtracking, buscamos alternativas que nos permitieran disminuirlo. Logramos realizar algunas mejoras, las cuales serán analizadas en Backtracking - Greedy. En ambos casos se realiza una poda cuando el algoritmo se da cuenta que la solución parcial no sirve. Esto ocurre cuando se hace una asignación de un maestro a un grupo que implica una suma mayor o igual a nuestra suma óptima. Cuando se da esta situación, se corta y se intenta asignarlo a otro grupo. Presentaremos a continuación ambas versiones del algoritmo.

3.2.1 Backtracking inicial

Este algoritmo de bactracking utiliza la siguiente función auxiliar para calcular la adición de los cuadrados de la suma de las habilidades de cada grupo:

```
47 def sumatoria(grupos):
48     suma = 0
49     for grupo in grupos:
50         suma_grupo = 0
51         for _maestro, habilidad in grupo:
52             suma_grupo += habilidad
```

```
53          suma += suma_grupo ** 2
54      return suma
```

Esta función recorre los k grupos para calcular la suma pedida. En el peor caso, la cantidad de maestros de un grupo puede ser n (con $k = 1$), por lo que tiene un costo proporcional a $n \cdot k$. Diremos entonces, que la complejidad temporal de esta porción de código es $O(n \cdot k)$, con $k \leq n$ puesto que si $k > n$, nuestro algoritmo completo se ejecuta en $O(1)$.

A continuación mostramos otras funciones que utiliza el código principal:

```
56  def caso_k_igual_a_n(maestros_y_habilidades):
57      suma = 0
58      grupos = []
59      for maestro, habilidad in maestros_y_habilidades:
60          grupos.append({maestro})
61          suma += habilidad**2
62      return grupos, suma
63
64  def obtener_resultado(S_con_habilidades):
65      resultado = []
66
67      for i in range(len(S_con_habilidades)):
68          grupo = set()
69          for maestro, _ in S_con_habilidades[i]:
70              grupo.add(maestro)
71          resultado.append(grupo)
72
73      return resultado
```

Como sabemos que si $n = k$ cada grupo tendrá únicamente un maestro, decidimos que en ese caso particular no se aplique el algoritmo de backtracking. En su lugar, resolvemos la asignación con la función `caso_k_igual_a_n` que itera sobre la lista de maestros, implicando un costo de $O(n)$. Esto nos permitió disminuir notablemente el tiempo de ejecución cuando $n = k$.

Por otro lado, como necesitamos la habilidad de cada maestro para poder minimizar la función objetivo, nuestro algoritmo tiene en cuenta grupos con el nombre y la fuerza de cada maestro. Luego los filtramos para quedarnos únicamente con los nombres. Éste es el objetivo de `obtener_resultado`. Su complejidad temporal es $O(n \cdot k)$, debido al mismo análisis que el de la función `sumatoria`.

Analizaremos ahora el código principal:

```
1   def problema_tribu_del_agua_bt(maestros_y_habilidades, k):
2       n = len(maestros_y_habilidades)
3
4       if k > n:
5           return None
```

```
6
7       if k == 0:
8           return [], 0
9
10      if k == n:
11          return caso_k_igual_a_n(maestros_y_habilidades)
12
13      S = [set() for _ in range(k)]
14      maestros_y_habilidades = sorted(maestros_y_habilidades,
15                                      key=lambda x: -x[1])
16
17      S_con_habilidades, coeficiente = problema_tribu_del_agua_bt_recur(
18          maestros_y_habilidades, 0, list(S), list(S), float('inf'))
19
20      resultado = obtener_resultado(S_con_habilidades)
21
22      return resultado, coeficiente
23
24
25  def problema_tribu_del_agua_bt_recur(maestros_y_habilidades,
26      indice_actual, S_actual, mejor_S, mejor_suma):
27
28      if indice_actual == len(maestros_y_habilidades):
29          suma_actual = sumatoria(S_actual)
30          if suma_actual < mejor_suma:
31              return list(map(set, S_actual)), suma_actual
32          return list(map(set, S_actual)), mejor_suma
33
34      for grupo in S_actual:
35          grupo.add(maestros_y_habilidades[indice_actual])
36          if sumatoria(S_actual) < mejor_suma:
37              nuevo_S, nueva_suma = problema_tribu_del_agua_bt_recur(
38                  maestros_y_habilidades, indice_actual+1, S_actual,
39                  mejor_S, mejor_suma)
40
41              if nueva_suma < mejor_suma:
42                  mejor_S, mejor_suma = nuevo_S, nueva_suma
43          grupo.remove(maestros_y_habilidades[indice_actual])
44
45      return mejor_S, mejor_suma
```

S $k > n$ no existe solución y si $k = 0$ no podemos formar grupos. En ambos casos el algoritmo funciona en $O(1)$. Sin embargo, se tratan de casos particulares. Procederemos a explicar el caso general.

En la `línea 14` ordenamos el conjunto de maestros por mayor habilidad con el algoritmo Timsort cuya complejidad es $O(n \cdot log(n))$, siendo n la cantidad de

maestros de la tribu que se enfrentarán a la Nación del Fuego. Crear los grupos vacíos en la línea `línea 13` es una operación lineal en la cantidad de grupos k, es decir, $O(k)$.

La solución del problema viene dada por la función recursiva `problema_tribu_del_agua_bt_recur`. Su objetivo es probar todas las combinaciones de asignaciones de maestros a los grupos de forma tal de minimizar la adición de los cuadrados de la suma de las habilidades de cada uno. Para cada maestro comprueba si, al asignarlo al grupo $i \in [0, k-1]$, se puede obtener una combinación con una suma menor a la actual. Caso contrario, poda y prueba con el siguiente grupo. Iniciamos con una suma con valor infinito. La cantidad de posibles asignaciones es k^n pues para cada uno de los n maestros hay k opciones de grupos. Por lo tanto, la complejidad de esta función y del algoritmo en general es exponencial, más específicamente $O(k^n)$.

3.2.2 Backtracking - Greedy

```
3  def problema_tribu_del_agua_bt_greedy(maestros_y_habilidades, k):
4      n = len(maestros_y_habilidades)
5
6      if k > n:
7          return None
8
9      if k == 0:
10         return [], 0
11
12     if k == n:
13         return caso_k_igual_a_n(maestros_y_habilidades)
14
15     maestros_y_habilidades = sorted(maestros_y_habilidades,
16                                     key=lambda x: -x[1])
17
18     S = [set() for _ in range(k)]
19     sumas_grupos = [0] * k #O(k)
20
21     S_con_habilidades, coeficiente = problema_tribu_del_agua_bt_greedy_recur(
22         maestros_y_habilidades, 0, list(S), list(S), float('inf'),
23         sumas_grupos)
24
25     resultado = obtener_resultado(S_con_habilidades)
26
27     return resultado, coeficiente
```

Esta función pública es bastante similar a la del algoritmo original, con la excepción de crear una lista de tamaño k, `sumas_grupos`, cuyo costo es lineal.

Ambas versiones ordenan los maestros por mayor habilidad. Observamos que ésto mejoró el tiempo de ejecución. No es greedy, pero es una optimización. Para

este caso decidimos seguir con una lógica similar, agregando otra optimización basada en el algoritmo de aproximación greedy propuesto por la cátedra. Éste consiste en asignar los maestros iterativamente en orden según su fuerza al grupo con la menor fuerza total hasta el momento. Se empieza por los más habilidosos. Utilizamos la misma idea para que cuando el algoritmo de backtracking intente agregar a un maestro a un grupo, lo haga en el grupo con menor habilidad. Si ésta no es una opción válida, continuamos con el siguiente grupo con menor fuerza. Ésto lo logramos ordenando los grupos. Como optimización adicional, guardamos en una lista la suma actual de cada grupo con el objetivo de evitar calcularlas en cada paso. La ordenamos en conjunto con los grupos usando Timsort, con un costo $O(k \cdot log(k))$.

```
29 def problema_tribu_del_agua_bt_greedy_recur(maestros_y_habilidades,
30     indice_actual, S_actual, mejor_S, mejor_suma, sumas_grupos):
31 
32     if indice_actual == len(maestros_y_habilidades):
33         suma_actual = sum(s ** 2 for s in sumas_grupos)
34         if suma_actual < mejor_suma:
35             return list(map(set, S_actual)), suma_actual
36         return list(map(set, S_actual)), mejor_suma
37 
38     S_actual, sumas_grupos = ordenar_por_suma(S_actual, sumas_grupos)
39     maestro = maestros_y_habilidades[indice_actual]
40 
41     for i, grupo in enumerate(S_actual):
42         grupo.add(maestro)
43         sumas_grupos[i] += maestro[1]
44         suma_actual = sum(s ** 2 for s in sumas_grupos)
45         if suma_actual < mejor_suma:
46             nuevo_S, nueva_suma = problema_tribu_del_agua_bt_greedy_recur(
47             maestros_y_habilidades, indice_actual+1, S_actual, mejor_S,
48             mejor_suma, sumas_grupos)
49 
50             if nueva_suma < mejor_suma:
51                 mejor_S, mejor_suma = nuevo_S, nueva_suma
52         sumas_grupos[i] -= maestro[1]
53         grupo.remove(maestro)
54 
55     return mejor_S, mejor_suma
56 
57 def ordenar_por_suma(S, sumas_grupos):
58     grupos_con_sumas = list(zip(S, sumas_grupos))
59 
60     grupos_con_sumas.sort(key=lambda x: x[1])
61 
62     S, sumas_grupos = zip(*grupos_con_sumas)
```

```
63
64      return list(S), list(sumas_grupos)
```

El algoritmo sigue realizando una búsqueda explícita del espacio de soluciones, por lo que su complejidad temporal es exponencial. Sin embargo, las mejoras nos permitieron reducir notablemente el tiempo de ejecución. Esto se debe a que se asigna rápidamente un maestro a un grupo con una suma pequeña, lo cual puede conducir a futuras podas (no se encuentra una suma mayor a la mejor hasta el momento). La comparación con el algoritmo original puede encontrarse en Backtracking - greedy.

3.3 Modelo de Programación Lineal

Debido a la dificultad de linealizar la función objetivo, nuestro modelo de programación lineal buscará, en cambio, minimizar la diferencia del grupo de mayor suma con el de menor suma. De esta forma, obtendremos una aproximación a la solución óptima. Sea Z el grupo con la mayor suma de habilidades de los maestros $\sum_i Z_i$ e Y el de menor suma, entonces se busca minimizar $\sum_i Z_i - \sum_j Y_j$.

Para resolver el problema utilizaremos programación lineal entera. A continuación detallaremos el modelo:

- Constantes → Las fuerzas de los maestros son constantes del problema. H_i es la habilidad del maestro i.
- Variables →
  - $X_{i,j}$: Es una variable binaria. Vale 1 si el maestro i es asigndo al grupo j, con $i \in [0, n-1]$ o $i \in [1, n]$ y $j \in [0, k-1]$ o $j \in [1, k]$. En caso contrario, vale 0.
  - S_{min}: Representa el valor del grupo con la menor suma. Sea S_j la suma del grupo j tal que $S_j = \sum_{i=1}^{n} H_i \cdot X_{i,j}$, entonces $S_{min} = \min_j S_j$.
  - S_{max}: Representa el valor del grupo con la mayor suma. $S_{max} = \max_j S_j$.
- Restricciones →
  - Asignaciones: cada maestro debe ser asignado a un solo un grupo. Luego, $sum_{j=1}^{k} X_{i,j} = 1 \quad \forall i$.
  - Sumas: $S_{min} \leq S_j \quad \forall j$ y $S_{max} \geq S_j \quad \forall j$.
- Función objetivo → se desea minimizar $S_{max} - S_{min}$.

Podemos observar la declaración de las variables, la función objetivo y las restricciones en el código:

```python
def problema_tribu_del_agua_pl(maestros_y_habilidades, k):
    num_maestros = len(maestros_y_habilidades)

    if k > num_maestros:
        return None

    if k == 0:
        return [], 0

    if k == num_maestros:
        return caso_k_igual_a_n(maestros_y_habilidades)

    # variables
    X = pulp.LpVariable.dicts("X", ((i, j)
    for i in range(num_maestros) for j in range(k)), cat='Binary')

    # suma de cada grupo
    S = [pulp.lpSum(maestros_y_habilidades[i][1] * X[i, j]
        for i in range(num_maestros)) for j in range(k)]

    # Variables: S_max = Z, S_min = Y
    S_max = pulp.LpVariable("S_max", lowBound=0, cat='Integer')
    S_min = pulp.LpVariable("S_min", lowBound=0, cat='Integer')

    problem = pulp.LpProblem("Problema_Tribu_del_Agua_con_PL",
                             pulp.LpMinimize)

    # Restricciones para las asignaciones: cada maestro debe
    # ser asignado a exactamente un grupo
    for i in range(num_maestros):
        problem += pulp.lpSum(X[i, j] for j in range(k)) == 1

    # Restricciones para S_max y S_min
    # S_max debe ser mayor o igual a todas las sumas
    # S_min debe ser menor o igual a todas las sumas
    for j in range(k):
        problem += S_max >= S[j]
        problem += S_min <= S[j]

    # Función objetivo: minimizar la diferencia entre el grupo
    # con mayor habilidad y el grupo con menor habilidad
    problem += S_max - S_min

    # Resolver
    problem.solve(pulp.PULP_CBC_CMD(msg=False))

```

```
50      return obtener_resultado(num_maestros, k,
51                               maestros_y_habilidades, X)
52
53  def obtener_resultado(num_maestros, k, maestros_y_habilidades, X):
54      resultado = [set() for _ in range(k)]
55      suma_por_grupo = [0 for _ in range(k)]
56
57      for i in range(num_maestros):
58          for grupo in range(k):
59              if pulp.value(X[i, grupo]) == 1:
60                  nombre = maestros_y_habilidades[i][0]
61                  habilidad = maestros_y_habilidades[i][1]
62                  resultado[grupo].add(nombre)
63                  suma_por_grupo[grupo] += habilidad
64
65      coeficiente = sum(s**2 for s in suma_por_grupo)
66      return resultado, coeficiente
67
68  # Cota set de ej adicionales: 1
69  # Cota set de mediciones: 1.0141856654211776
70  # Cota set de la cátedra:
```

Definimos los mismos casos particulares que en los otros algoritmos, con el objetivo de disminuir el tiempo de ejecución cuando conocemos el resultado del problema.

Crear las variables $X_{i,j}$ y calcular las sumas implican, para cada grupo, iterar sobre los n maestros que pueden ser asignados al mismo. Por lo tanto, cada una de estas operaciones conlleva $O(k \cdot n)$. Con el objetivo de definir la restricción de las asignaciones, se obtiene, para cada maestro, todas las variables asociadas según el grupo. Luego, ésto también es $O(n \cdot k)$. En cambio, las restricciones para S_{min} y S_{max} implican iterar por los k grupos, con un costo $O(k)$.

La función `obtener_resultado` tiene dos ciclos anidados que realizan operaciones constantes para cada maestro según el grupo. La creación de las listas de `resultado` y `suma_por_grupo`, así como el cálculo del coeficiente, son operaciones lineales en la cantidad de grupos. Entonces, la complejidad temporal de la función es $T(n) = 3 \cdot O(k) + O(n \cdot k) = O(n \cdot k)$.

Por último, la resolución del algoritmo utilizando la biblioteca `pulp` consume tiempo exponencial porque se trata de programación lineal entera.

3.4 Algoritmos de Aproximación

3.4.1 Aproximación de la cátedra

3.4.2 Aproximación adicional

La siguiente aproximación propuesta sigue una lógica similar a la sugerida por la cátedra, pero con un tiempo de ejecución menor.

```python
def problema_tribu_del_agua_aprox_adicional(maestros_y_habilidades, k):
    n = len(maestros_y_habilidades)

    if k > n:
        return None

    if k == 0:
        return [], 0

    if k == n:
        return caso_k_igual_a_n(maestros_y_habilidades)

    S = [set() for _ in range(k)]

    maestros_y_habilidades = sorted(maestros_y_habilidades,
                                    key=lambda x: -x[1])

    grupo_actual = 0
    sumas = [0] * k
    for maestro_y_habilidad in maestros_y_habilidades: #O(n)
        if grupo_actual == k:
            grupo_actual = 0

        nombre, habilidad = maestro_y_habilidad

        S[grupo_actual].add(nombre)
        sumas[grupo_actual] += habilidad
        grupo_actual += 1

    coeficiente = sum(s**2 for s in sumas) #O(k)
    return S, coeficiente
```

Nuevamente ordenamos los maestros por habilidad descendiente, con un costo $O(n \cdot log(n))$. Luego, este algoritmo sigue la regla sencilla de asignar de forma iterativa el maestro con la mayor habilidad al siguiente grupo en una secuencia cíclica, es decir, el maestro más habilidoso estará en el grupo 1, el k en el grupo k y el $k+1$ nuevamente en el primer grupo. El objetivo es distribuir los maestros de mayor habilidad primero y de manera uniforme a través de todos los grupos, lo cual evita la acumulación de habilidades altas en un solo grupo. El algoritmo es greedy porque sigue la estrategia mencionada para obtener un óptimo local en cada paso, dado por el grupo actual según el ciclo, con la esperanza de encontrar una solución globalmente óptima. Sin embargo, como el problema es NP-Completo y la solución exacta conlleva un costo exponencial, podemos afirmar que este algoritmo no es óptimo, sino tan solo una aproximación.

La asignación de los grupos implica iterar sobre los n maestros, lo cual es

lineal, $O(n)$. Por otro lado, calcular el coeficiente utilizando la lista auxiliar `sumas` conlleva $O(k)$. El resto de las operaciones son constantes. En total, la complejidad del algoritmo propuesto, para el caso general, es

$$\mathcal{T}(n) = \mathcal{O}(n \cdot log(n)) + \mathcal{O}(n) + \mathcal{O}(k) = \mathcal{O}(n \cdot log(n))$$

3.5 Efecto de las variables sobre el algoritmo

Todos los algoritmos propuestos consideran los siguientes casos

- $k > n$: No existe solución. Devolvemos `None` en $O(1)$.
- $k = 0$: Todos los algoritmos se ejecutan en $O(1)$, devolviendo una lista vacía y coeficiente 0.
- $k = n$: Cada grupo tendrá un solo maestro. Lo resolvemos en tiempo lineal $O(n)$.
- $k < n$: Es el caso general y la complejidad depende del algoritmo utilizado. Los algoritmos exactos conllevan tiempo exponencial, aumetando considerablemente con el valor de k y n.

4 Ejemplos de ejecución

En la carpeta `ejemplos_adicionales` se pueden encontrar distintos casos de prueba que agregamos con el objetivo de comprobar la correctitud de los algoritmos propuestos. A continuación detallamos cada uno:

- `uno_por_grupo.txt` → En este caso $k = n$, por lo que cada maestro será asignado a un grupo distinto.
- `habilidades_similares.txt` → Los maestros tienen habilidades distintas, pero parejas.
- `habilidades_ascendentes.txt` → Las tuplas de maestros vienen ordenadas ascendentemente según la fuerza.
- `habilidades_descendentes.txt` → Las tuplas de maestros vienen ordenadas descendentemente según la fuerza.
- `habilidades_iguales.txt` → Los maestros tienen la misma habilidad.
- `grupos_parejos.txt` → Las habilidades de los maestros son tales que, al realizar la asignación, cada grupo tendrá la misma suma.
- `una_habilidad_alta.txt` → Uno de los maestros tiene una habilidad muy alta en comparación con la del resto.
- `k_menor_a_n.txt` → Caso general cuando $k < n$.

De forma adicional, agregamos la posibilidad de ejecutar los ejemplos utilizados para las mediciones. Los mismos se encuentran en `ejemplos_mediciones`. Como siempre, en `ejemplos_catedra` tenemos los casos de prueba provistos por la cátedra.

4.1 Ejecución del programa

En esta sección explicaremos las distintas formas de ejecutar el programa.

- `python3 codigo/main.py`: ejecutará todos los casos de prueba existentes y mostrará los grupos formados, el coeficiente resultante y el tiempo de ejecución para cada algoritmo. Se ejecutan los ejemplos adicionales, los de las mediciones y los provistos por la cátedra. En el caso de backtracking solo se ejecuta la versión mejorada.
- `python3 codigo/main.py ruta_a_ejemplo`: procesará los datos del archivo dado y ejecutará todos los algoritmos, mostrando el resultado de cada uno, así como su tiempo de ejecución.
- `python3 codigo/main.py ruta_a_ejemplo --flag`: ejecutará el algoritmo según el flag utilizado. Si es inválido, por defecto actúa como el anterior.

  `--bt` → Backtracking, versión inicial

  `--btg` → Backtracking, versión mejorada

  `--pl` → Programación Lineal

  `--a1` → Algoritmo de aproximación propuesto por la cátedra

  `--a2` → Algoritmo de aproximación adicional
- `python3 codigo/main.py --flag`: Ejecuta todos los tests usando el algoritmo propuesto. Si es inválido, por defecto se comporta como si no tuviera el flag.
- `python3 codigo/main.py N`: Ejecuta los tests usando todos los algoritmos, pero limita los ejemplos de la cátedra a N. El objetivo es poder ejecutar la mayor cantidad de tests en un tiempo menor.
- `python3 codigo/main.py N --flag`: Se comporta como el anterior, pero solo ejecuta los tests con el algoritmo indicado.

Recomendamos utilizar:

- `python3 codigo/main.py ruta_a_ejemplo`
- `python3 codigo/main.py N --flag`

4.2 Cotas de aproximación empírica

Mediante el uso de todos los ejemplos de ejecución logramos determinar de forma empírica la cota de los algoritmos de aproximación.

Sea I una instancia cualquiera del problema, $z(I)$ una solución óptima para dicha instancia y sea $A(I)$ la solución aproximada, se define $\frac{A(I)}{z(I)} \leq r(A)$ para todas las instancias posibles. Calculamos la cota $r(A)$ obteniendo la máxima razón $\frac{A(I)}{z(I)}$ de entre todos los casos de prueba.

Los resultados obtenidos son los siguientes:

- Programación lineal $\rightarrow$ $r(A) = 1.0141856654211776 \approx 1.01419$. En general, los resultados coincidieron con el óptimo. En otros casos, la diferencia entre las soluciones resultó ser muy pequeña.

- Aproximación adicional $\rightarrow$ Para los sets de datos de la cátedra, los usados en las mediciones y los adicionales, $r(A) = 1.2157676348547717 \approx 1.21577$. También realizamos mediciones con volúmenes de datos elevados para el algoritmo exacto, con el fin de obtener más información de la cota. Probamos casos en los que sabemos que nuestro algoritmo no es óptimo. Por ejemplo, si $k = 2$ y se tiene una habilidad muy alta y el resto bastante pequeñas. Llegamos hasta $n = 100000$. Los ejemplos se pueden encontrar en `codigo/cota_aprox_adicional.py`. El resultado obtenido fue $r(A) = 1.2999951999748 \approx 1.3$.

5 Mediciones de tiempo

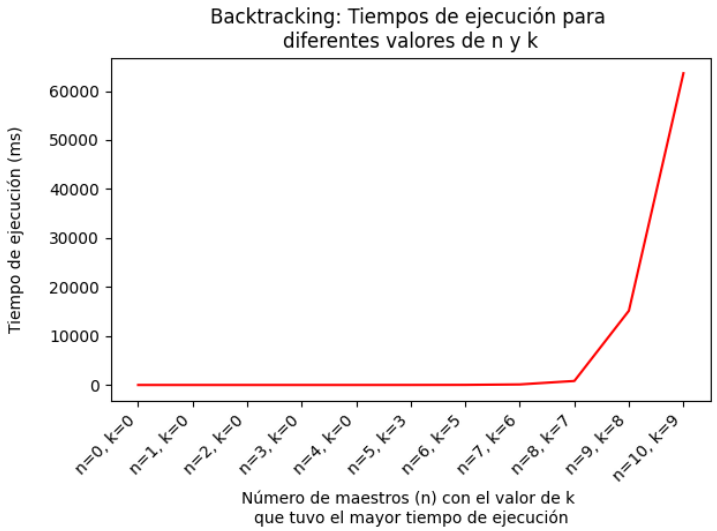
Para corroborar la complejidad algorítmica de los algoritmos implementados, realizamos una serie de tests. Probamos distintas combinaciones de n y k, para $n \in [0, 10]$ y $k \in [0, n]$. Para cada n tomamos en cuenta la combinación de n y k con mayor tiempo de ejecución. Tomamos estas mediciones en todos los casos para realizar comparaciones. Para los algoritmos de aproximación también se tomaron mediciones adicionales con valores de n y k más elevados.

El código que usamos para generar los tests se encuentran en el archivo `codigo/grafico_complejidad.py`.

5.1 Algoritmo de backtracking

5.1.1 Backtracking inicial

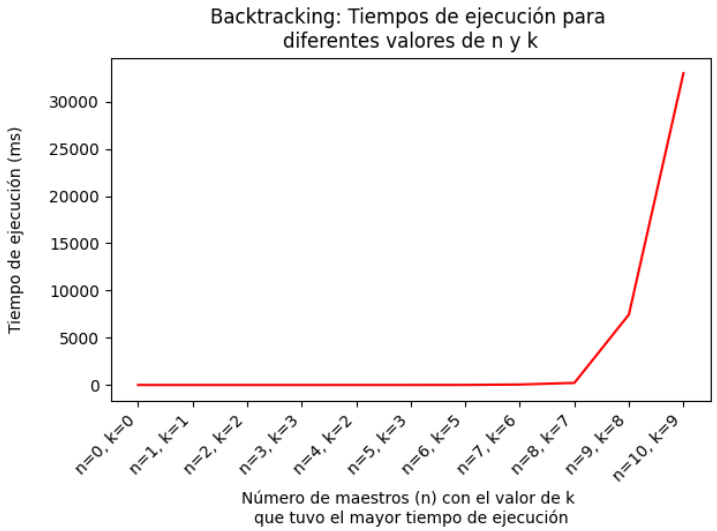
Con este algoritmo de backtracking obtuvimos los siguientes resultados:



Como podemos observar en la tendencia de la curva, el tiempo de ejecución aumenta exponencialmente con la cantidad de maestros n y también depende de k. El tiempo más alto ocurre cuando k se acerca a n, con $k < n$ (es lineal cuando $k = n$). Para valores pequeños de estas variables el algoritmo es relativamente rápido. Sin embargo, al incrementarlos no se vuelve práctico debido a que el tiempo no crece polinomialmente, sino exponencialmente. Esto corrobora el análisis de la complejidad planteado previamente.

5.1.2 Backtracking - greedy

Procederemos a comparar esta versión del algoritmo de backtrcking con la anterior.

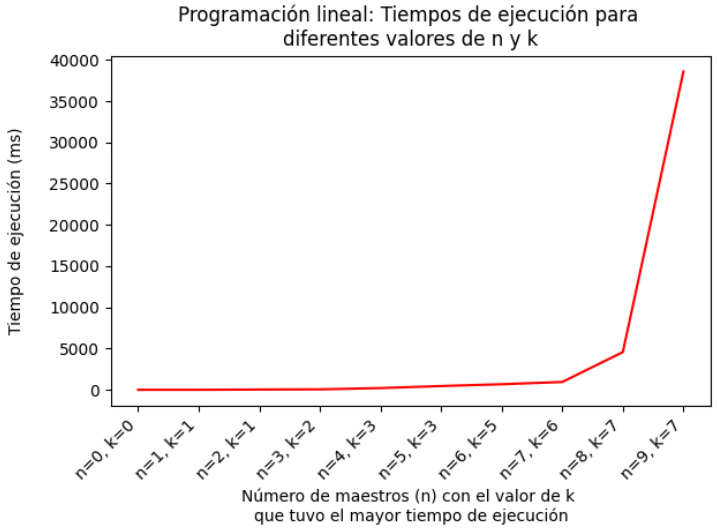


El gráfico evidencia una drástica disminución de los tiempos de ejecución, casi a la mitad de los del otro algoritmo.

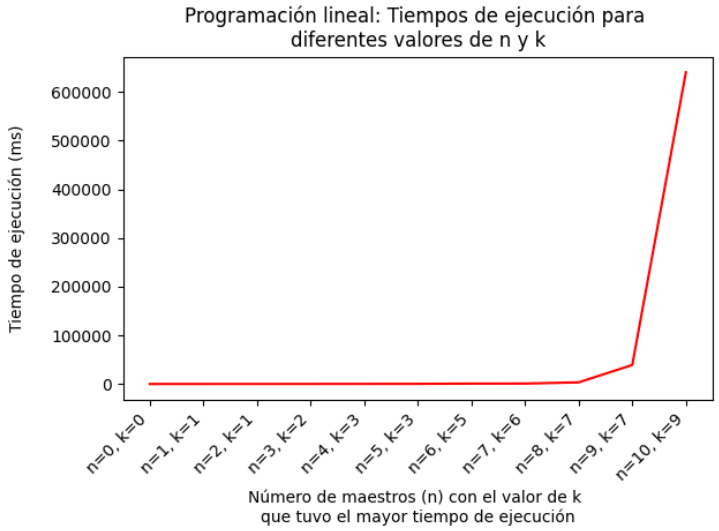
Nuevamente podemos observar la tendecia exponencial de la curva, que comprueba nuestra justificación de la complejidad temporal.

5.2 Algoritmo de programación lineal

Tomamos mediciones de tiempo con los mismos sets de datos utilizados para backtracking. Realizamos 2 gráficos para facilitar la comparación entre algoritmos.



El mejor algoritmo de backtracking tarda aproximadamente 35000 milisegundos en ejecutarse en el peor caso para $n = 10$. Sin embargo, PLE ya necesita 40000 milisegundos para $n = 9$. Esto es una gran diferencia con ambos algoritmos de backtracking que conllevan menos de 20000 milisegundos en ese caso.



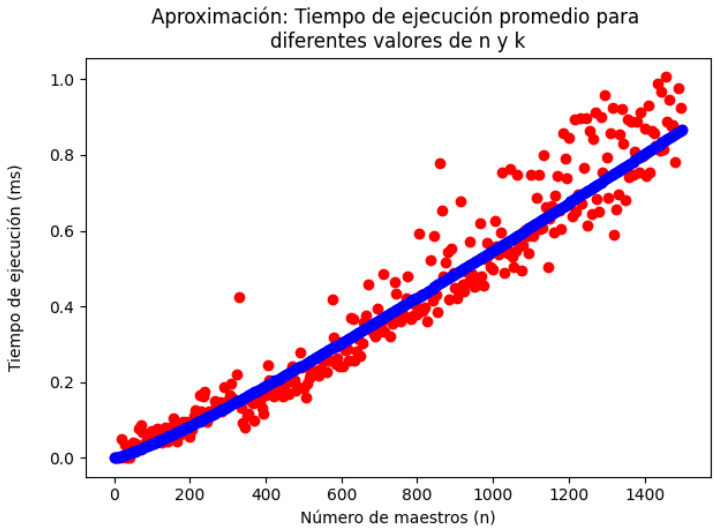
Analizando la totalidad de las mediciones, es evidente que nuestro algoritmo de programación lineal obtuvo tiempos de ejecución significativamente mayores a las dos implementaciones de backtracking. En todos los casos la complejidad temporal es exponencial, lo cual puede observarse en los respectivos gráficos. No obstante, para el conjunto de datos de entrada dado, el desempeño del algoritmo de PLE es peor.

5.3 Algoritmos de Aproximación

5.3.1 Aproximación de la cátedra

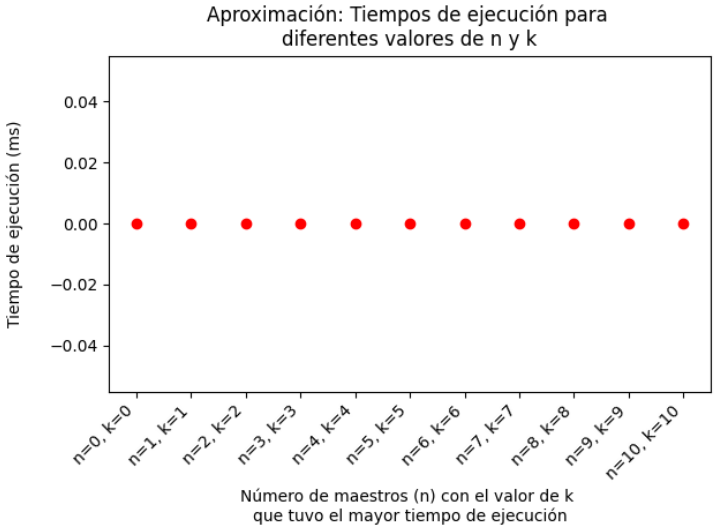
5.3.2 Aproximación adicional

Realizamos mediciones con n y k hasta 1500 para corroborar la complejidad algorítmica de esta aproximación con valores inmanejables para el algoritmo exacto.



Se puede observar una curva azul que representa $n \cdot log(n)$, mientras que los puntos rojos son nuestras mediciones.

También realizamos un gráfico con el set de datos usado en backtracking y programación lineal para mostrar la rapidez de este algoritmo en comparación con los exponenciales.



6 Conclusión