Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos, Backtracking y Programación Lineal

13 de junio de 2024

Tomás Hevia
110934

Manuel Campoliete
110479

Andrés Colina
110680

1. Introducción

Dado el Problema de la Tribu del Agua en su versión de optimización, se busca encontrar la solución óptima mediante un algoritmo de Backtracking y aproximarse a la solución mediante un Modelo de Programación Lineal y un algoritmo de aproximación Greedy. Luego se le realizarán mediciones a los algoritmos planteados, para poder compararlos y relacionarlos entre sí (tanto su tiempo de ejecución como el nivel de aproximación a la solución óptima, en el caso de los algoritmos de aproximación). Por otra parte, se tiene la versión de decisión del problema, para la cual se demostrará que el mismo es NP-Completo, determinando previamente que efectivamente está en NP.

1.1. Problema de la Tribu del Agua

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

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

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

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

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

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

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

1.1.1. Versión de Decisión

Dado una secuencia de n fuerzas/habilidades de maestros de agua $x_1, x_2, ..., x_n$, y dos números k y B, definir si existe una partición en k subgrupos $S_1, S_2, ..., S_k$ tal que:

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

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

2. Demostración NP-Completo

2.1. ¿El algoritmo esta en NP?

Antes de demostrar que el problema es NP-Completo, debemos demostrar que esta en NP. Esto lo demostramos mediante un verificador eficiente que valide la solución al problema en tiempo polinomial

```python
def validador(maestros,k,grupos_sol,minimo,B):
    if len(grupos_sol) != k: # O(1)
        return False

    if minimo > B: # O(1)
        return False

    if len(maestros) != sum(len(grupo) for grupo in grupos_sol): # O(k)
        return False

    for maestro in maestros:
        esta_en_sol = False
        for i in range(k):
            if maestro in grupos_sol[i]:
                esta_en_sol = True
                break
        if not esta_en_sol:
            return False

    return True
```

La complejidad del validador es $O(n * k)$, siendo n la cantidad de maestros y k la cantidad de grupos. Entonces **podemos afirmar que el Problema del Agua está en NP**.

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

Para demostrar que el Problema del Agua es NP-Completo, se debe reducir un problema NP-Completo al Problema del Agua, habiendo demostrado previamente que el Problema del Agua está en NP.

Para ello, vamos a reducir el problema de 2-Partition al Problema de la Tribu del Agua.

2.2.1. Definición del Problema de 2-Partition

El problema de 2-Partition en su versión de problema de decisión sería: Dado un Conjunto de n elementos ¿Es posible dividir el conjunto en dos subconjuntos tales que cada elemento del conjunto original se encuentre en uno de los dos subconjuntos y que ambos tengan la misma suma? (*)

(*) El anexo 9.1 contiene la demostración de que 2-Partition es un problema NP-Completo.

2.2.2. 2-Partition $\leq_p$ Problema de la Tribu del Agua

Sea el conjunto A definido como:

$$A = \{x_1, x_2, \ldots, x_n\}$$

Se busca dividirlo en dos subconjuntos A_1 y A_2 tal que la suma de sus valores sean iguales.

Para resolver 2-Partition mediante el Problema de la Tribu del Agua, se deben transformar los datos de entrada de 2-Partition para que sean compatibles y puedan ser usados como datos de entrada del Problema de la Tribu del Agua.

Como datos de entrada del Problema de la Tribu del Agua necesitamos un conjunto de n valores, un valor k y un valor B tales que:

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

- El conjunto A de n valores se usa como el conjunto de n fuerzas/habilidades de los maestros del agua.
- k es igual a 2. Con esto buscamos que los valores se dividan en 2 subconjuntos.
- B es igual a $2S^2$, siendo S la mitad de la suma total de los elementos de A. Al dividir los valores en dos subconjuntos, la sumatoria de cada subconjunto debe ser S para que el 2-Partition se resuelva. Dentro del Problema de la Tribu del Agua, la sumatoria de los valores de cada grupo se eleva al cuadrado y se suman entre si, teniendo asi $S^2 + S^2 = 2S^2$, que es el valor al cual se debería llegar para que 2-Partition sea verdadero.

Si hay Tribu del Agua $\rightarrow$ hay 2-Partition

Si existe una partición de los maestros en 2 grupos tales que la suma objetivo es menor o igual que B, como $B = 2S^2$, eso necesariamente implica que la suma de ambos grupos es S. Supongamos la suma de los cuadrados de ambos grupos es $2S^2$ pero que la suma de los grupos por separado no es S. Entonces la diferencia entre ambos es de un valor $0 < A < 2S$, es decir la suma del mayor grupo al cuadrado será $(S + A)^2$ y la suma del menor grupo al cuadrado sera $(S - A)^2$

Desarrollando:

$$\begin{aligned}(S + A)^2 + (S - A)^2 &= S^2 + 2SA + A^2 + S^2 - 2SA + A^2 \\ &= S^2 + A^2 + S^2 + A^2 \\ &= 2S^2 + 2A^2\end{aligned}$$

Hemos llegado a una contradicción, pues la suma de ambos grupos es mayor que $2S^2$. Por lo tanto, la suma necesariamente debe dar $2S^2$, entonces cada grupo suma S y por ende existe un 2-Partition.

Si hay 2-Partition $\rightarrow$ hay Tribu del Agua

Si se cumple 2-Partition, hay dos subconjuntos del conjunto original que suman por separado S. Luego el problema de la Tribu del Agua va a encontrar esta partición (pues solo se puede dividir en dos grupos y cumple que la suma de éstos al cuadrado es justamente $2S^2$) y además no va a poder encontrar una cuya suma sea menor por lo demostrado en el punto anterior.

Habiendo demostrado la reducción, como 2-Partition es a lo sumo tan difícil como el problema de la Tribu del Agua y 2-Partition es un problema NP-Completo, como el problema de la Tribu del Agua está en NP, por transitividad el **problema de la Tribu del Agua es NP-Completo**.

Queda demostrado: **2-Partition $\leq_p$ Problema de la Tribu del Agua**

3. Algoritmo de Backtracking

```
1  IDX_HAB = 1 # Usamos esta constante para todos los algoritmos
2
3  def bt(maestros,k,estan_ordenados,minimo_greedy=float('inf')):
4      maestros_ord = maestros
5      if not estan_ordenados:
6          maestros_ord=sorted(maestros,key=lambda maestro: maestro[IDX_HAB],reverse=
       True)
7      grupos_aux = [set() for _ in range(k)]
8      grupos_sol = [set() for _ in range(k)]
9      suma_grupos = [0 for _ in range(k)]
10     minimo = [minimo_greedy]
11
12     _bt(maestros_ord,k,0,grupos_aux,grupos_sol,suma_grupos,minimo,0)
13     return minimo[0],grupos_sol
14
15 def actualizar(grupos_sol, grupos_aux):
16     for i in range(len(grupos_aux)):
17         grupos_sol[i].clear()
18         grupos_sol[i].update(grupos_aux[i])
19
20 def _bt(maestros,k,i,grupos_aux,grupos_sol,suma_grupos,minimo,cuadrados):
21     if i == len(maestros):
22         if cuadrados <= minimo[0]:
23             minimo[0] = cuadrados
24             actualizar(grupos_sol,grupos_aux)
25         return
26
27     if cuadrados >= minimo[0] or suma_grupos.count(0)>len(maestros)-i:
28         return
29
30     for j in range(k):
31         grupos_aux[j].add(maestros[i])
32         nueva_suma_grupo = suma_grupos[j] + maestros[i][IDX_HAB]
33         nuevos_cuadrados = cuadrados - (suma_grupos[j]**2) + (nueva_suma_grupo**2)
34         suma_grupos[j] = nueva_suma_grupo
35         _bt(maestros,k,i+1,grupos_aux,grupos_sol,suma_grupos,minimo,
       nuevos_cuadrados)
36         grupos_aux[j].remove(maestros[i])
37         suma_grupos[j] -= maestros[i][IDX_HAB]
38         if suma_grupos[j] == 0:
39             return
```

A continuación una breve mención de las podas realizadas:

- La obvia: si la suma actual supera el mínimo parcial obtenido, podamos.
- Si la cantidad de grupos vacíos supera la cantidad de maestros que falta asignar a un grupo, podamos. Con esta poda, evitamos explorar soluciones que no tienen sentido para el problema.
- Si al quitar un maestro de un grupo ese grupo quedó vacío, podamos, pues no queremos probar colocar a dicho maestro en el siguiente grupo (también vacío). De ser así estaríamos repitiendo en distinto orden, la misma combinatoria (y además estaríamos dejando un grupo vacío).

Queremos mencionar también la optimización realizada sobre el cálculo de los cuadrados. Durante la implementación del algoritmo, nos hemos dado cuenta que al agregar un maestro a un grupo, sólo se ve modificado el cuadrado de la suma de ese grupo en cuestión y que por lo tanto no es necesario calcular, en cada llamado, la suma de los cuadrados de la suma de los grupos. Esto se evidencia entre las líneas 32 y 36, siendo esta última la del llamado recursivo con el cálculo de los nuevos cuadrados. Por otro lado también se notó una gran mejora en los tiempos de ejecución al ordenar los maestros por mayor habilidad, se asume que esto sucede gracias a que se llega más rápido a un mejor mínimo. Por último se usa como mínimo inicial el encontrado por la aproximación Greedy para así hacer que la primera poda mencionada sea mucho más efectiva.

La complejidad del algoritmo de Backtracking resulta $O(k^n)$ siendo n la cantidad de maestros y k la cantidad de grupos, pues probamos la asignación de cada uno de los maestros a cada uno de los k grupos.

4. Resolución por Programación Lineal

Con el objetivo de minimizar la diferencia del grupo de mayor suma con el de menor suma, se plantea el siguiente Modelo de Programación Lineal, conformado por las variables $X_{i,j}$, una serie de restricciones y la función objetivo a minimizar. Mediante este modelo se obtiene una aproximación al resultado óptimo del Problema de la Tribu del Agua.

4.1. Modelo de Programación Lineal Entera

Constantes para el problema:

- x_i (habilidad del maestro i)

Variables:

- $X_{i,j}$ variable booleana (el maestro i está en el grupo j)

Restricciones:

- $\sum_i X_{i,j} = 1, \quad \forall i$ (cada maestro debe estar asignado a un grupo y sólo un grupo)
- $\sum_j X_{i,j} \geq 1, \quad \forall j$ (cada grupo debe tener al menos un maestro asignado, no puede haber grupos vacíos)
- $\sum_i X_{i,j_{\max}} \cdot x_i \geq \sum_i X_{i,j} \cdot x_i, \quad \forall j \neq j_{\max}$ (*) (hay un grupo Z cuya suma de habilidades es mayor a la de los demas grupos)
- $\sum_i X_{i,j_{\min}} \cdot x_i \leq \sum_i X_{i,j} \cdot x_i, \quad \forall j \neq j_{\min}$ (*) (hay un grupo Y cuya suma de habilidades es menor a la de los demas grupos)

(*) Fijamos arbitrariamente $j_{\max}$ en 0 y $j_{\min}$ en k-1, lo cual es indistinto. Podríamos haber elegido otros índices.

Función Objetivo a Minimizar:

- $\sum_i Z_i - \sum_i Y_i = \sum_i X_{i,j_{\max}} \cdot x_i + \sum_i X_{i,j_{\min}} \cdot x_i$

4.2. Algoritmo de PL

Para desarrollar este modelo, se utiliza la libreria Pulp de Python, la cual permite plantear y resolver Modelos de Programación Lineal.

```
import pulp

def pl(maestros, k):

    n = len(maestros)
    x_i = [maestro[IDX_HAB] for maestro in maestros]

    vars = []
    for i in range(n):
        fila_vars = []
        for j in range(k):
            nombre_var = "X" + str(i) + str(j)
            fila_vars.append(pulp.LpVariable(nombre_var, cat="Binary"))
```

```
        vars.append(fila_vars)

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

    for i in range(n):
        fila_vars = vars[i]
        problem += pulp.LpAffineExpression([(var, 1) for var in fila_vars]) == 1

    ecuaciones = []

    for j in range(k):
        columna_vars = []
        for i in range(n):
            columna_vars.append(vars[i][j])
        problem += pulp.LpAffineExpression([(var, 1) for var in columna_vars]) >= 1
        ecuaciones.append(pulp.LpAffineExpression([(var, x_i[i]) for i, var in
    enumerate(columna_vars)]))

    i = 1
    while i < k:
        problem += ecuaciones[0] >= ecuaciones[i]
        i += 1
    i = 0
    while i < k - 1:
        problem += ecuaciones[k-1] <= ecuaciones[i]
        i += 1

    problem += (ecuaciones[0] - ecuaciones[k-1])

    problem.solve()

    values = []
    for i in range(n):
        values.append(list(map(lambda Xij: pulp.value(Xij), vars[i])))

    minimo = 0
    grupos = []
    for j in range(k):
        suma_grupo = 0
        grupo=set()
        for i in range(n):
            suma_grupo += values[i][j] * x_i[i]
            if values[i][j]:
                grupo.add(maestros[i])
        minimo += suma_grupo ** 2
        grupos.append(grupo)

    return int(minimo), grupos
```

La complejidad del algoritmo es exponencial, por ser Programación Lineal Entera (branch and bound).

4.3. Relación entre Programación Lineal y Backtracking

En pos de obtener una cota de aproximación empírica, comparamos los mínimos obtenidos por el algoritmo de Programación Lineal con los resultados óptimos obtenidos por el algoritmo de Backtracking.

Definimos:

- La solución de Backtracking (óptima) como $z(I)$
- La solución por Programación Lineal (aproximación) como $A(I)$
- La cota de aproximación como $\frac{A(I)}{z(I)} \leq r(A)$ para toda instancia I del problema

n	$A(I)$	$z(I)$	$\frac{A(I)}{z(I)}$
4	1332642	1332642	1
5	1970926	1970926	1
6	4278266	4278266	1
7	2182815	2182815	1
8	2613921	2610987	1.001
9	5995185	5993077	1.0003
10	6273483	6273483	1
11	7400426	7400426	1
12	7773266	7773266	1
13	7678766	7678766	1
14	12809254	12809254	1
15	14394452	14394452	1
16	19833663	19833663	1
17	15657866	15657866	1
18	16491722	16491722	1

Cuadro 1: Cantidad $k = 4$ de grupos fija, Cantidad n de guerreros variable

k	$A(I)$	$z(I)$	$\frac{A(I)}{z(I)}$
2	24703421	24703421	1
3	37241634	37241634	1
4	12348202	12348202	1
5	19626052	19626012	1,000002

Cuadro 2: Cantidad k de grupos variable, cantidad $n = 17$ de guerreros fija

Entonces, podemos tomar $1{,}001 \leq r(A)$

5. Algoritmo Greedy Recomendado

El algoritmo recomendado por la cátedra para poder lograr una aproximación al problema de la Tribu del Agua es el siguiente: primero generamos los k grupos vacíos. Luego ordenamos de mayor a menor los maestros en función de su habilidad o fortaleza. Agregamos al maestro más habilidoso al grupo con menos habilidad hasta ahora (cuadrado de la suma). Repetimos siguiendo con el siguiente más habilidoso, hasta que no queden más maestros por asignar.

```
def greedy(maestros, k, estan_ordenados):
    maestros_ord = maestros
    if not estan_ordenados:
        maestros_ord = sorted(maestros, key=lambda maestro: maestro[IDX_HAB],
    reverse=True)
    grupos = [set() for _ in range(k)]

    i = 0
    while i < k:
        grupos[i].add(maestros_ord[i])
        i += 1

    while i < len(maestros_ord):
        grupo_min = min(grupos, key=lambda grupo: sum(hab for (_, hab) in grupo) **
     2)
        grupo_min.add(maestros_ord[i])
        i += 1

    return sum(sum(hab for (_, hab) in grupo) ** 2 for grupo in grupos), grupos
```

Es un algoritmo Greedy pues aplica una regla sencilla para obtener el óptimo local al estado

actual del algoritmo: en cualquier iteración arbitraria del algoritmo asignamos el maestro con mayor habilidad (aún no asignado a ningún grupo) al grupo con menor habilidad hasta ahora. Aplicamos dicha regla iterativamente esperando obtener el óptimo global: la mejor distribución posible de maestros en los grupos tal que se minimice la sumatoria de los cuadrados de la suma de los grupos.

La complejidad de este algoritmo es $O(nlogn)$, dado que se realiza un ordenamiento sobre el arreglo de maestros y posteriormente se itera el mismo.

5.1. Relación entre el Algoritmo Greedy y el Backtracking

De forma análoga a la comparación de los coeficientes obtenidos entre el algoritmo de Backtracking y el algoritmo de Programación Lineal, realizamos lo mismo con el algoritmo de aproximación Greedy.

Idénticamente: sea I una instancia cualquiera del problema, y $z(I)$ 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

Definimos:

- La solución de Backtracking (óptima) como $z(I)$
- La solución de Greedy (aproximación) como $A(I)$
- La cota de aproximación como $\frac{A(I)}{z(I)} \leq r(A)$ para toda instancia del problema

n	$A(I)$	$z(I)$	$\frac{A(I)}{z(I)}$
4	1332642	1332642	1
5	1970926	1970926	1
6	4278266	4278266	1
7	2182815	2182815	1
8	2615787	2610987	1.001
9	6004715	5993077	1.001
10	6274169	6273483	1.0001
11	7428186	7400426	1.003
12	7778158	7773266	1.0006
13	7681330	7678766	1.0003
14	12811244	12809254	1.0001
15	14408414	14394452	1.0009
16	19834535	19833663	1.00004
17	15673534	15657866	1.001
18	16492774	16491722	1.00006

Cuadro 3: Cantidad k = 4 de grupos fija, Cantidad n de guerreros variable

k	$A(I)$	$z(I)$	$\frac{A(I)}{z(I)}$
2	24705665	24703421	1.00009
3	37260538	37241634	1.0005
4	12360808	12348202	1.001
5	19681606	19626012	1.002
6	15572044	15540208	1.002
7	11251468	11226264	1.002
8	10855337	10849969	1.0004
9	8632869	8632869	1
10	9180665	9180665	1

Cuadro 4: Cantidad k de grupos variable, Cantidad n = 17 de guerreros fija

Entonces, podemos tomar 1,003 $\leq r(A)$

Además, analizamos los tests más grandes proveídos por la cátedra de los cuales se conoce su valor óptimo, con el fin de corroborar la cota empírica calculada con nuestros tests.

n_k	$A(I)$	$z(I)$	$\frac{A(I)}{z(I)}$
18_6	10325588	10322822	1.0003
18_8	12000279	11971097	1.0024
20_4	21083935	21081875	1.0001
20_5	16838539	16828799	1.0006
20_8	11423826	11417428	1.0006

Efectivamente, se corrobora la cota obtenida.

6. Algoritmo Greedy Extra

Proponemos el siguiente algoritmo de aproximación extra. Sigue la misma lógica que el algoritmo propuesto por la cátedra, pero en vez de asignar el maestro con mayor habilidad (aún no asignado) al grupo de menos habilidad, asigna al grupo con menor cantidad de maestros, esperando así también minimizar la sumatoria propuesta al distribuir uniformemente los maestros, no por habilidades pero sí por cantidad.

```
1  def greedy2(maestros, k, estan_ordenados):
2
3      maestros_ord = sorted(maestros, key=lambda maestro: maestro[IDX_HAB], reverse=
       True)
4      grupos = [set() for _ in range(k)]
5
6      for maestro in maestros_ord:
7          grupo_menor_cantidad=min(grupos, key=lambda grupo: len(grupo))
8          grupo_menor_cantidad.add(maestro)
9
10     return sum(sum(hab for _, hab in grupo) ** 2 for grupo in grupos), grupos
```

Para obtener el óptimo local, este algoritmo Greedy usa como regla el tomar el maestro con mayor habilidad y ponerlo en el grupo con menor cantidad de maestros, así busca obtener el óptimo local y, aplicando la regla iterativamente, lograr aproximarse al óptimo global del problema. Este algoritmo no es óptimo, pero logra una buena aproximación al problema. La complejidad de este algoritmo es $O(nlogn)$

6.1. Relación entre el Algoritmo Greedy Extra y el Backtracking

De forma análoga a la comparación de los coeficientes obtenidos entre el algoritmo de Backtracking y el algoritmo de Aproximación Greedy, realizamos lo mismo con el algoritmo Greedy

Extra.

Sea I una instancia cualquiera del problema, y $z(I)$ 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

Definimos:

- La solución de Backtracking (óptima) como $z(I)$
- La solución de Greedy (aproximación) como $A(I)$
- La cota de aproximación como $\frac{A(I)}{z(I)} \leq r(A)$ para toda instancia del problema

n	$A(I)$	$z(I)$	Relación
4	1332642	1332642	1
5	2129290	1970926	1.1
6	4467214	4278266	1.04
7	2503363	2182815	1.1
8	2873305	2610987	1.1
9	6457151	5993077	1.1
10	6808993	6273483	1.1
11	7605774	7400426	1.03
12	8054044	7773266	1.04
13	7991062	7678766	1.04
14	13159226	12809254	1.03
15	14702206	14394452	1.02
16	20195561	19833663	1.02
17	15938742	15657866	1.02
18	16623052	16491722	1.01

Cuadro 5: cantidad k de grupos fija, cantidad n de guerreros variable

k	$A(I)$	$z(I)$	Relación
2	24820065	24703421	1.005
3	37509716	37241634	1.01
4	12777906	12348202	1.03
5	19941976	19626012	1.02
6	15937248	15540208	1.03
7	11766874	11226264	1.05
8	11446427	10849969	1.06
9	9418817	8632869	1.1
10	10065373	9180665	1.1

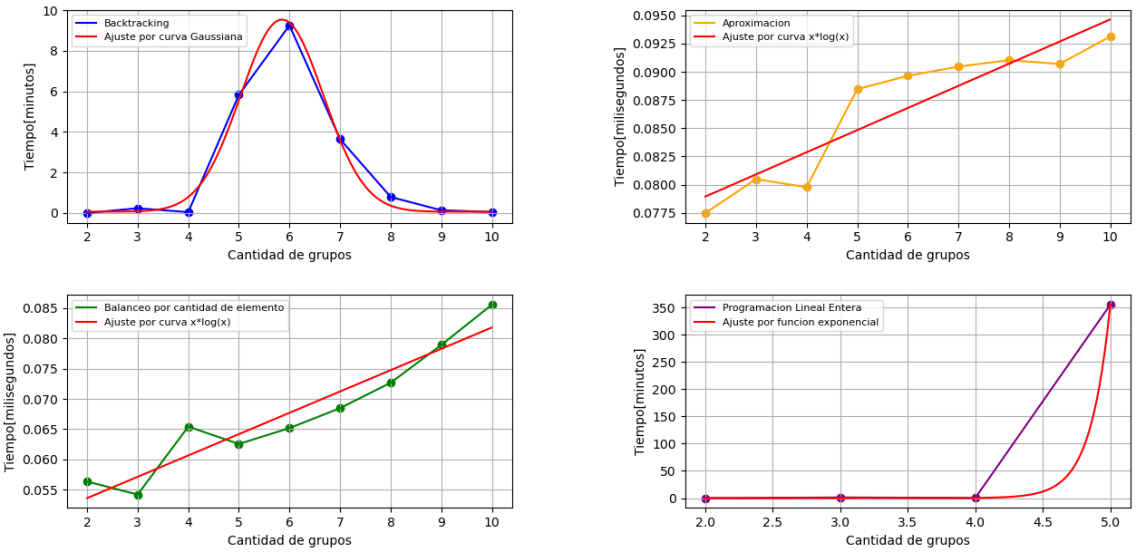
Cuadro 6: Cantidad k de grupos variable, cantidad n de guerreros fija

Entonces, podemos tomar 1,1 $\leq r(A)$

7. Mediciones

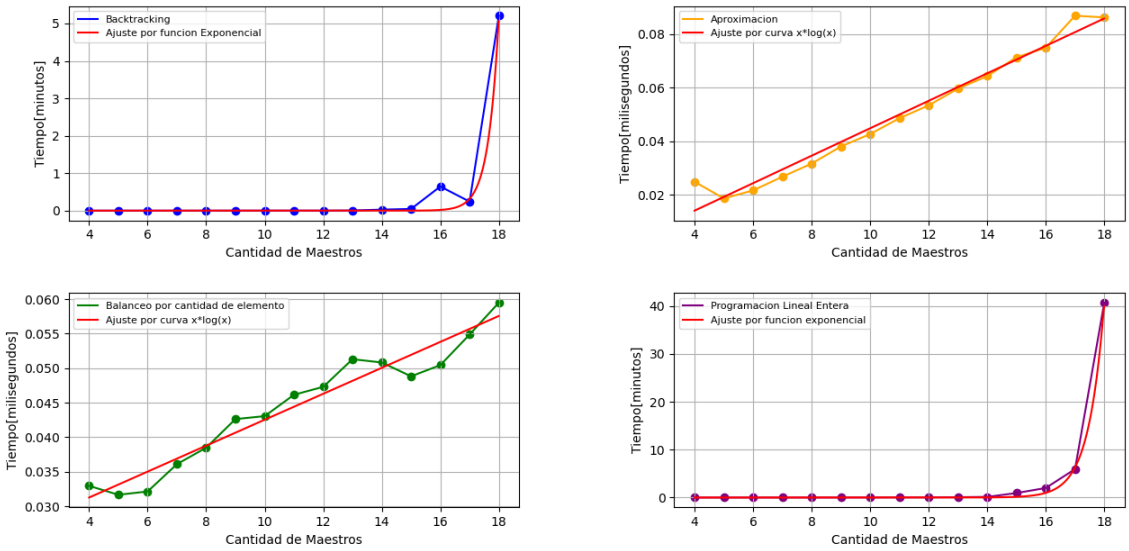
Para obtener las mediciones correspondientes se armaron dos sets de datos distintos, en uno variamos la cantidad de grupos y dejamos la cantidad de maestros fija (17 maestros, y la cantidad de grupos van del 2 al 10) y para el otro la misma lógica pero invertida (se mantienen 4 grupos y se varía la cantidad de maestros de 4 a 18 maestros). Para ambos casos, el número representativo de la habilidad de cada maestro se generó con el módulo **random** de Python, entre 1 y 1000. Nota: Para generar los gráficos se usó el módulo **Mathplotlib** y para buscar la curva que mejor se ajusta a los datos se usó la función **curve-fit** del módulo **Scipy.optimize**

7.1. Mediciones con cantidad de Maestros fija



- Se observa que curiosamente el algoritmo de Backtracking tiene un alto desempeño para un número de grupos bajo y alto, lo que genera una forma de Campana de Gauss (lo cual es evidente pues la mejor curva que lo ajusta es una función Gaussiana). Se intuye que este comportamiento es debido a la tercera poda mencionada en el apartado **3**, pues el algoritmo se ahorra recorrer ramas que en sí son sólo combinaciones dependientes del orden en que se van uniendo maestros en cada grupo.
- Ambos gráficos correspondientes a los algoritmos Greedy tienen una tendencia compatible a la complejidad esperada (la curva $x*log(x)$ hallada se ajusta muy bien a los datos obtenidos). Es destacable que, como era esperable, para el mismo set de datos el algoritmo de Backtracking tarda mas que ambos Greedy, con la excepción de valores de K pequeños y grandes donde el algoritmo de Backtracking se destaca y no se tarda mucho mas que ambos Greedy.
- Para el Caso de Programación Lineal Entera, las mediciones que fueron posibles de tomar (pues el algoritmo, aun siendo una aproximación, es descomunalmente lento) se observa claramente un comportamiento exponencial. Notar que hasta el caso de 4 grupos el algoritmo era razonable pero para 5 grupos crece el tiempo de ejecución explosivamente. Es destacable que aun tratándose de una aproximación, el algoritmo es **mucho más lento** que el algoritmo de Backtracking implementado (lo cual muestra la eficiencia de este último).

7.2. Mediciones con cantidad de grupos Fija



- En este caso, el algoritmo de Backtracking presenta una crecimiento exponencial, lo cual es esperado dada la complejidad predicha.
- Se mantienen los comentarios dichos en el apartado **7.1** para los gráficos Greedy, con la salvedad de que esta vez el algoritmo de Backtracking no puede competir contra estos en tiempo de ejecución para mayor volumen de elementos.
- Si bien el algoritmo de Programación Lineal Entera sigue teniendo un comportamiento exponencial, es destacable que para el caso en cuestión se desempeña mejor que en el expuesto en el apartado **7.1**, es decir el algoritmo es mucho mas eficiente para casos donde se varia la cantidad de maestros (siempre y cuando la cantidad de grupo no sea muy grande). No obstante, el algoritmo sigue siendo mucho mas lento que el Backtracking implementado.

8. Conclusión

Luego de un arduo análisis se remarca que el Problema de la Tribu del Agua es en definitiva un problema NP-Completo. Dada la dificultad del problema es conveniente analizarlo usando diferentes técnicas algorítmicas para poder entenderlo en profundidad. Viendo las mediciones tomadas es claro que si bien el algoritmo implementado por Backtracking siempre da la solución óptima, para una gran cantidad de maestros y una cantidad de grupos "media", el algoritmo se vuelve exponencialmente más lento. Por ello nos es útil pensar en aproximaciones que si bien no siempre nos dan una solución óptima, son sobresalientes por su velocidad de ejecución. Luego, nos resulta interesante notar que el modelo de Programación Lineal Entera tiene entre todas las aproximaciones la mejor cota de todas, pero irónicamente tiene un tiempo de ejecución peor que el propio algoritmo de Backtracking, por lo que el modelo no resulta de utilidad alguna para el caso de estudio. Para finalizar podemos confirmar que gracias a la recomendación del maestro Pakku la Tribu del Agua logra defenderse con éxito del ataque de la Nación del Fuego.

9. Anexos

9.1. Subset-Sum $\leq_p$ 2-Partition

Sea (L, N) una instancia del problema de la Subset Sum, donde L es una lista de números, y N es la suma objetivo. Sea $S = \sum L$. Sea L' la lista formada agregando los valores $(S + N)$ y $(2S - N)$ a L. Observamos que $\sum L' = 4S$.

$(\implies)$

Si existe un sublista $M \subseteq L$ que suma N, entonces L' puede ser particionada en dos partes iguales: $M \cup \{2S - N\}$ y $L \setminus M \cup \{S + N\}$. De hecho, la primera parte suma $N + (2S - N) = 2S$, y la segunda suma $(S - N) + (S + N) = 2S$.

$(\impliedby)$

Si L' puede ser particionado en dos partes iguales P_1, P_2, entonces probaremos que hay un sublista L que suma N.

Recordemos que, dado que $\sum L' = 4S$ entonces $\sum P_1 = \sum P_2 = 2S$.

Notar que $(S + N) + (2S - N) = 3S \neq 2S$, por lo tanto, estos dos elementos se encuentran en lados opuestos de la partición.

Sin pérdida de generalidad, supongamos que $2S - N \in P_1$. Por lo tanto, el resto de P_1 debe venir exactamente de L y debe sumar a N, por lo que hemos encontrado un sublista que suma N, como se necesitaba.

9.2. Tiempos de Ejecución de los Tests de la Cátedra con Backtracking y Greedy

Mencionamos, anecdóticamente, los tiempos que obtuvimos con el algoritmo de Backtracking (una sola pasada) para los tests proveídos por la cátedra. Cabe destacar que estas pasadas fueron realizadas en una máquina distinta a la utilizada para tomar el resto de las mediciones hechas. Sólo las mostramos para que se vea aproximadamente los tiempos que obtuvimos para dichos tests.

Archivo	Tiempo
5_2	0.07 ms
6_3	0.11 ms
6_4	0.15 ms
8_3	0.34 ms
10_3	5.00 ms
10_5	7.88 ms
10_10	0.17 ms
11_5	4.19 ms
14_3	348.76 ms
14_4	644.92 ms
14_6	1.28 s
15_4	479.71 ms
15_6	2.19 s
17_5	1 m 40 s
17_7	4.18 s
17_10	1.47 s
18_6	6 min 10 s
18_8	27 min 30 s
20_4	32 min 12 s
20_5	109 min
20_8	16 min