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 Complejidad algorítmica

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  def generarTestDe(archivo):
10     k = None
11     maestros_y_habilidades = []
12
13     with open(archivo, "r") as file:
14         for i, line in enumerate(file):
15             line = line.strip()
16             if line.startswith("#"):
17                 continue
18             if k is None:
19                 k = int(line)
20                 continue
21             maestro, habilidad = line.split(", ")
22             maestros_y_habilidades.append((maestro, int(habilidad)))
23
24     return maestros_y_habilidades, k
```

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 Complejidad algoritmo de Backtracking

Nuestro 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 $O(1)$.

Analizaremos ahora el código principal:

```
1  def problema_tribu_del_agua_bt(maestros_y_habilidades, k):
2      if k > len(maestros_y_habilidades):
```

```
        return None

    if k == 0:
        return [], 0

    S = [set() for _ in range(k)]
    maestros_y_habilidades = sorted(maestros_y_habilidades,
                                    key=lambda x: -x[1])
    S_con_habilidades, coeficiente = problema_tribu_del_agua_bt_recur(
        maestros_y_habilidades, k, 0, list(S), list(S), float('inf'))

    resultado = []

    for i in range(len(S_con_habilidades)):
        grupo = set()
        for maestro, _ in S_con_habilidades[i]:
            grupo.add(maestro)
        resultado.append(grupo)

    return resultado, coeficiente


def problema_tribu_del_agua_bt_recur(maestros_y_habilidades,
    k, indice_actual, S_actual, mejor_S, mejor_suma):

    if indice_actual == len(maestros_y_habilidades):
        suma_actual = sumatoria(S_actual)
        if suma_actual < mejor_suma:
            return list(map(set, S_actual)), suma_actual
        return list(map(set, S_actual)), mejor_suma

    for grupo in S_actual:
        grupo.add(maestros_y_habilidades[indice_actual])
        if sumatoria(S_actual) < mejor_suma:
            nuevo_S, nueva_suma = problema_tribu_del_agua_bt_recur(
                maestros_y_habilidades, k, indice_actual+1, S_actual,
                mejor_S, mejor_suma)

            if nueva_suma < mejor_suma:
                mejor_S, mejor_suma = nuevo_S, nueva_suma
        grupo.remove(maestros_y_habilidades[indice_actual])

    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 9` 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 8` es una operación lineal en la cantidad de grupos k, es decir, $O(k)$. En cambio, recorrer la solución obtenida por la función recursiva para quedarnos únicamente con los nombres de los maestros de cada grupo implica un costo $O(n \cdot k)$ porque para cada asignación tenemos un `for` anidado que la itera, realizando operaciones constantes adicionales.

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 $O(k^n)$.

3.3 Complejidad algoritmo de Programación Lineal

3.4 Complejidad algoritmo de Aproximación

3.5 Efecto de las variables sobre el algoritmo

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`. Por defecto, al ejecutar todos los tests solo se ejecutan aquellos en los que $k \neq 0$, $k \leq 9$ y $n \leq 10$.

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.
- `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

  `--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:

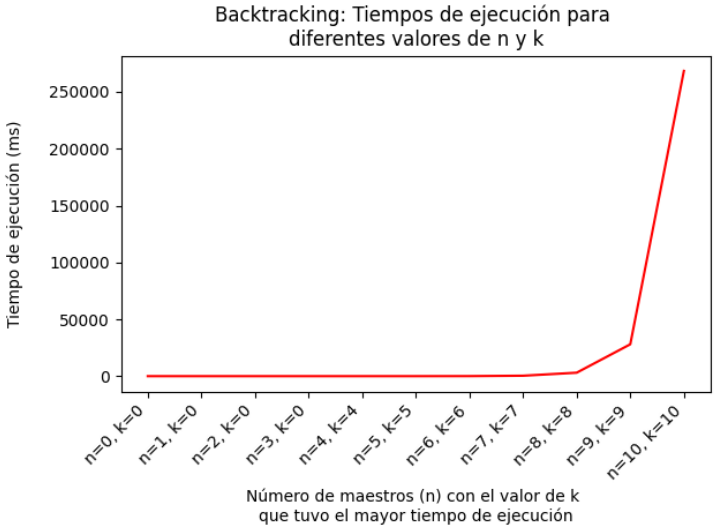
- `python3 codigo/main.py ruta_a_ejemplo`
- `python3 codigo/main.py N --flag`

5 Mediciones de tiempo

Para corroborar la complejidad algorítmica de los algoritmos implementación, 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. El código que usamos para generar los tests se encuentran en el archivo `codigo/grafico_complejidad.py`.

5.1 Algoritmo de backtracking

Con el 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. 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.

5.2 Algoritmo de programación lineal

5.3 Algoritmo de aproximación

6 Conclusión