.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
La mafia de los algortimos greedy

10 de abril de 2025

Antonella Briglia
90903

Germán Bobadilla
90123

1. Análisis del problema

El objetivo es determinar si es posible hacer coincidir cada timestamp s_i con exactamente un intervalo $[t_i - e_i, t_i + e_i]$ de tal forma que s_i pertenezca al intervalo. Si todos los timestamp del sospechoso coinciden con un intervalo de las transacciones sospechosas, y cada intervalo es usado como máximo una vez, entonces el sospechoso es culpable.

1.1. Propuesta del algoritmo

La idea principal del algoritmo propuesto es ordenar los intervalos por su extremo derecho $(t_i + e_i)$ ya que al elegir el primer intervalo que termina más pronto, dejamos una mayor cantidad de espacio disponible para emparejar los siguientes intervalos.

Esta elección es una típica estrategia *Greedy* porque en cada paso se está eligiendo la decisión óptima local, la cual nos lleva a una solución óptima global para todas las coincidencias.

Los pasos del algoritmo son:

1. Se generan los n intervalos de las transacciones sospechosas a partir de los valores t_i y e_i.
2. Se ordenan los intervalos por su extremo derecho $(t_i + e_i)$ en orden ascendente.
3. Se recorren los timestamps s_i del sospechoso (asumiendo que vienen ordenados).
4. Se recorren también los intervalos.
5. Para cada timestamp s_i, se busca el primer intervalo que:
   - No haya sido utilizado previamente.
   - Contenga el valor s_i dentro de su rango.
6. Si se encuentra un intervalo compatible, se guarda la coincidencia y se marca ese intervalo como usado.
7. Si no se encuentra un intervalo válido para algún s_i, el sospechoso no es la rata.

2. Demostración

Como explicamos en el punto anterior, el algoritmo propuesto se basa en un enfoque *Greedy* que ordena los intervalos por su extremo derecho y, para cada timestamp del sospechoso, elige el primer intervalo válido disponible (aquel que no fue usado y contiene al timestamp). Queremos demostrar que:

- Si hay coincidencia, el algoritmo la encuentra.
- Si no existe una asignación válida, el algoritmo lo detecta.

Para determinar la correspondencia entre las transacciones con intervalos y los timestamps sospechosos se debe cumplir:

- Cada timestamp sospechoso debe caer dentro de un intervalo.
- Cada uno de los intervalos debe ser usado una única vez.

Si hay coincidencia, el algoritmo la encuentra: Para cada uno de los movimientos sospechosos el algoritmo recorre los intervalos ordenados por su extremo derecho y selecciona el primero que no haya sido utilizado previamente y que el movimiento sospechoso esté incluido en ese intervalo de tiempo.

Esta estrategia es correcta porque al ordenar por el extremo derecho estamos eligiendo el intervalo que nos maximiza las posibilidades de que los próximos timestamps también encuentren un intervalo válido. Al ser Greedy, en cada paso realiza esta misma estrategia sin bloquear innecesariamente intervalos futuros.

Si no existe una asignación válida, el algoritmo lo detecta: Si para algún timestamp del sospechoso no se encuentra un intervalo que lo contenga y que no haya sido previamente utilizado, el algoritmo termina inmediatamente e informa que no se puede hacer una correspondencia completa. En este caso no existe una asignación válida posible porque el algoritmo ya probó todos los candidatos posibles para ese timestamp sospechoso.

¿Por qué sabemos que esta estrategia es óptima? Este problema tiene similitudes con el clásico problema del *scheduling* de charlas, donde también se ordenan las charlas por sus tiempos de finalización y se asignan secuencialmente para evitar solapamientos. Este problema está ampliamente estudiado y demostrado que admite una solución óptima mediante un enfoque Greedy, como el que usamos acá.

3. Algoritmo propuesto

Luego de armar los intervalos en la variable `transacciones`, ordenamos estos por su extremo derecho:

```
# Ordenamos los intervalos por su extremo derecho
transacciones.sort(key=lambda x: x[1])
```

Por otro lado, en la variable `sospechoso` tenemos las transacciones del sospechoso, las cuales asumimos que ya vienen ordenadas.

Finalmente se corre el código que plasma el algoritmo Greedy propuesto:

```
coincidencias = []
usados = set()  # Para no reutilizar intervalos

j = 0  # Indice para las transacciones del sospechoso
while j < n:
    i = 0 # Indice de las transacciones candidatas (intervalos)
    encontrado = False
    while i < n:
        inicio, fin, intervalo = transacciones[i]
        if i not in usados and inicio <= sospechoso[j] <= fin:
            coincidencias.append((intervalo, sospechoso[j]))
            usados.add(i)
            encontrado = True
            break  # Se encontr  una coincidencia, se pasa a la siguiente
    transacci n del sospechoso
        i += 1

    if not encontrado:
        print("No es el sospechoso correcto")
        return

    j += 1
```

En la variable `coincidencias` quedarán almacenados los datos pedidos, que luego se imprimirán siempre y cuando se haya encontrado a la rata.

3.1. Análisis de complejidad

Analicemos la complejidad del algoritmo: Tenemos n transacciones sospechosas y n transacciones con intervalos.

- **Construcción de intervalos:** Se recorren las n líneas del archivo con los pares (t_i, e_i) para construir los intervalos $[t_i - e_i, t_i + e_i]$. Esto cuesta $\mathcal{O}(n)$.
- **Ordenamiento de los intervalos:** Se ordenan los n intervalos por su tiempo de finalización $(t_i + e_i)$. Esto cuesta $\mathcal{O}(n \log n)$.
- **Lectura de los n timestamps del sospechoso:** Como ya vienen ordenados, simplemente se leen. Esto cuesta $\mathcal{O}(n)$.
- **Asignación de timestamps del sospechoso a los intervalos:** Para cada uno de los n timestamps sospechosos, se recorre la lista de intervalos (hasta n en el peor caso) buscando el primero que lo contenga y no haya sido usado. Esto, en el peor caso, implica una búsqueda $\mathcal{O}(n)$ por cada timestamp, resultando en $\mathcal{O}(n^2)$ en total.

Complejidad total:

- Construcción de intervalos: $\mathcal{O}(n)$
- Ordenamiento: $\mathcal{O}(n \log n)$

- Búsqueda de coincidencias: $\mathcal{O}(n^2)$
- $\Rightarrow$ Complejidad total: $\mathcal{O}(n^2)$ en el peor caso.

Aunque la complejidad es $\mathcal{O}(n^2)$ en el peor caso, en la práctica la ejecución puede ser más rápida dependiendo de cómo se distribuyen y solapan los intervalos. Por ejemplo:

- **Intervalos muy amplios:** Aumentan la cantidad de candidatos válidos para cada s_j, haciendo que el algoritmo deba revisar más intervalos hasta encontrar uno disponible, acercándose al peor caso $\mathcal{O}(n^2)$.
- **Intervalos más cortos o poco solapados:** Reducen el número de candidatos compatibles con cada s_j, lo que hace que las búsquedas sean más rápidas en promedio.
- **Timestamps del sospechoso que caen fuera de la mayoría de los intervalos:** El algoritmo puede detectar rápidamente que no hay coincidencia, terminando antes y reduciendo el tiempo total.

4. Ejemplos de ejecución

Además de los ejemplos provistos por la cátedra realizamos las siguientes ejecuciones

1. **Todos los intervalos se solapan parcialmente**

   Este caso testea que el algoritmo elija de manera greedy el mejor ajuste, sin que los solapamientos generen conflictos.

   `4-es.txt`

   ```
   4
   100,50
   130,50
   160,50
   190,50
   120
   140
   170
   180
   ```

   El algoritmo greedy debería asignar el primer 120 al de menor extremo derecho (150), y así sucesivamente.

   Resultado:

   ```
   120 --> 100 ± 50
   140 --> 130 ± 50
   170 --> 160 ± 50
   180 --> 190 ± 50
   ```

2. **Varias coincidencias pero con una sola combinación válida**

   Este test verifica que si hay varias coincidencias posibles, el algoritmo elija bien para que todas las transacciones coincidan.

   `3-es.txt`

   ```
   3
   100,30
   130,30
   160,30
   120
   160
   140
   ```

   Si el algoritmo asigna 120 a 130 ± 30, después no puede meter 140. En cambio, si asigna 120 a 100 ± 30, 140 a 130 ± 30 y 160 a 160 ± 30, sí entra todo.

   Resultado:

   ```
   120 --> 100 ± 30
   160 --> 130 ± 30
   140 --> 160 ± 30
   ```

3. **Sin coincidencia en una sola transacción**

   Para testear que el algoritmo falle cuando hay un solo timestamp fuera de rango, aunque los demás coincidan.

   `5-no-es.txt`

   ```
   5
   100,10
   200,10
   300,10
   400,10
   500,10
   105
   205
   305
   405
   999
   ```

   Las primeras 4 encajan perfecto, pero 999 no entra en ningún intervalo.

   Resultado:

   ```
   No es el sospechoso correcto
   ```

4. **Mismos timestamps en distinto orden**

   Esto testea que el algoritmo no se base en el orden de entrada de los intervalos, sino en el orden de los extremos derechos como dicta el greedy.

   `3-es-bis.txt`

   ```
   3
   160,30
   100,30
   130,30
   120
   140
   160
   ```

   Aunque el primero que aparece es 160,30, el algoritmo debe ordenarlos por el extremo derecho.

   Resultado:

   ```
   120 --> 100 ± 30
   140 --> 130 ± 30
   160 --> 160 ± 30
   ```

5. Mediciones de tiempo

Como mencionamos anteriormente la complejidad total del algoritmo es $\mathcal{O}(n^2)$ en el peor caso.

Para realizar esta verificación, se procedió de la siguiente manera:

- Se construyeron casos de prueba con una cantidad de transacciones cada vez mayor.
- Se ejecutó el algoritmo muchas veces para cada tamaño, promediando los tiempos obtenidos con el fin de reducir el ruido en la medición.
- Se utilizó un modelo de regresión por mínimos cuadrados para ajustar una función de la forma $T(n) = a \cdot n^2 + b$, correspondiente a una complejidad $O(n^2)$.
- Se realizó un gráfico comparando los tiempos promedio medidos con la curva ajustada, para observar qué tan bien se corresponden.

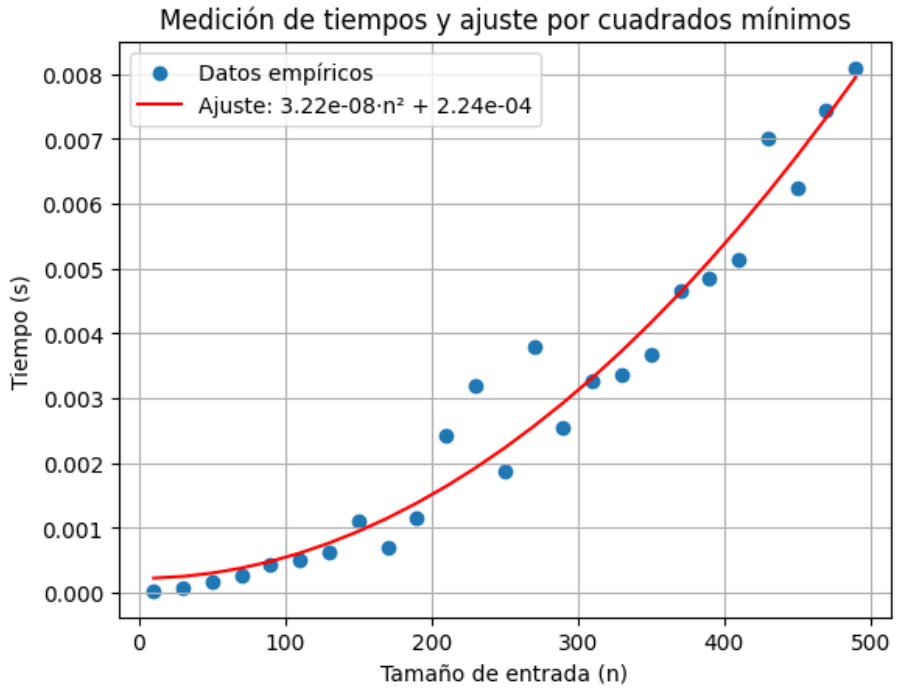

Figura 1: Tiempos promedio de ejecución y ajuste cuadrático.
Código fuente disponible en `github.com/germanbc/TDA/...`

El ajuste por mínimos cuadrados muestra que los tiempos de ejecución siguen una función cuadrática con buena precisión, confirmando que la complejidad empírica del algoritmo es $O(n^2)$, tal como se esperaba según el análisis teórico.

6. Conclusiones

A lo largo del desarrollo de este trabajo, exploramos diversas versiones del algoritmo para resolver el problema de coincidencia entre los timestamps sospechosos y los intervalos generados a partir de las transacciones. En las primeras implementaciones, detectamos que ciertos conjuntos de datos no eran correctamente resueltos, ya sea por una asignación incorrecta de los intervalos o por una lógica que no contemplaba todos los casos borde. Esto nos llevó a revisar tanto el enfoque como los criterios de ordenamiento y selección de intervalos.

Finalmente, llegamos a una versión final que ordena los intervalos por su tiempo de finalización más corto y asigna a cada timestamp el primer intervalo disponible que lo contenga. Esta solución no solo logró pasar correctamente todos los casos de prueba proporcionados por la cátedra, sino que también resolvió exitosamente los casos diseñados por nosotros mismos para cubrir escenarios más complejos. Además, el análisis empírico y teórico coincidieron en que la complejidad del algoritmo es $O(n^2)$.