Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy

18 de septiembre de 2023

Agostina Picchetti - 108028

Nehuén Cabibbo - 108025

Roberta Sprenger - 104239

1. Introducción

Greedy es una técnica de diseño de algoritmos que construye la solución a un problema paso a paso, aplicando iterativamente una regla sencilla que permite obtener una solución óptima local con la intención de que la sucesión de óptimos locales lleve a la mejor solución global.

En el presente trabajo practico buscamos aplicar esta técnica algorítmica para resolver óptima mente el problema planteado: minimizar el tiempo que le llevara al director técnico y a su equipo de trabajo analizar los próximos rivales de la Selección Argentina.

2. Análisis del problema y diseño del algoritmo

Tras evaluar y analizar el problema, decidimos tratarlo como una variante de un problema de *scheduling*, donde hay n tareas que deben realizarse (el análisis de cada compilado), pero el tiempo de cada una de esas tareas depende de dos actores, Scaloni (con sus tiempos representados como s_i) y alguno de los n ayudantes disponibles (con sus tiempos representados como a_i) y el objetivo es optimizar el tiempo de completitud de todas las tareas.

Observamos que :

- Scaloni necesariamente tiene que ver todos los videos y, por lo tanto, hará su análisis secuencialmente.
- De la observación anterior podemos deducir que el orden en que Scaloni vea los videos es irrelevante. El tiempo total que le lleve analizar lo n rivales será siempre el mismo, independientemente del orden en que lo haga.
- No podemos optimizar el tiempo total del análisis de rivales intentando mejorar el tiempo de la tarea de Scaloni, porque ese tiempo necesariamente va a transcurrir. Sin embargo, podemos inferir que sí puede minimizarse paralelizando lo más posible el trabajo de los n ayudantes para que la mayor cantidad de ellos realice su análisis durante el intervalo de tiempo que le lleva a Scaloni realizar el suyo. Si llamamos S a la suma de todos los s_i, el tiempo total del análisis de rivales será: $S +$ *tiempo excedente de los ayudantes*.
- Intuitivamente, entendemos que el orden de los compilados que minimiza el tiempo total de análisis se obtendrá disponiéndolos decrecientemente por su correspondiente a_i.
- Este orden de asignación de los compilados permite que los ayudantes que más tiempo demoran, al comenzar primeros, puedan trabajar en simultáneo con Scaloni y aún sobrepasando su tiempo, es el caso donde menos se excederían y por lo tanto donde menos sumarían al tiempo total de análisis. De esta forma se logra llegar a una solución óptima, minimizando el tiempo que lleva analizar los n rivales.

2.1. Análisis de la elección greedy

Hecho el estudio del problema, para diseñar nuestro algoritmo descartamos las siguientes opciones:

- Ordenar por s_i: Además de lo mencionado anteriormente respecto a la imposiblidad de optimizar los tiempos de trabajo de Scaloni, usando el ejemplo provisto por el curso de tres elementos, es evidente que no conviene ordenar por este criterio porque si lo hiciéramos, el tiempo total del análisis sumaría diecisiete, lejos del valor óptimo que es diez.
- Ordenar de mayor a menor por la suma $s_i + a_i$: Evaluamos también como alternativa ordenar por el tiempo de cada análisis, contemplando el que le lleva a Scaloni y el que le lleva al ayudante asignado, pero esta opción también falla en su optimalidad en casos como el siguiente:

s_i	a_i
1	8
5	2
3	3

El cálculo del tiempo total da doce, cuando el óptimo es once.

- Ordenar de menor a mayor por la relación s_i/a_i: Otra de las opciones que contemplamos fue ordenar por relación entre s_i y a_i y, si bien dió resultados correctos cuando probamos con sets de tres elementos, fallaba para sets de mayor cantidad, donde no daba exacto, pero sí una buena aproximación. Puede verse en el siguiente contraejemplo, que el resultado no es óptimo:

s_i	a_i
1	4
4	8
1	1

El cálculo del tiempo total da trece, cuando el óptimo es doce.

Finalmente, la elección *Greedy* para obtener el tiempo óptimo, será siempre quedarnos primero con el compilado del ayudante de a_i mayor. Para ello, ordenamos los compilados de rivales de mayor a menor según su a_i y luego, sobre ese orden, buscamos iterativamente el compilado con el máximo tiempo de finalización.

Entonces, proponemos dos funciones:

```
def calcular_tiempo_total_de_analisis(videos_rivales):
  fin_scaloni = 0
  fin_analisis = 0
  for video in videos_rivales:
    fin_scaloni += video["S_i"]
    fin_ayudante = fin_scaloni + video["A_i"]
    if fin_analisis < fin_ayudante:
      fin_analisis = fin_ayudante
  return fin_analisis
```

```
def ordenar_rivales(videos_rivales):
  videos_ordenados = sorted(videos_rivales, key=lambda t:t["A_i"], reverse=True)
  return videos_ordenados
```

Que usamos en el algoritmo *Greedy* que devuelve los compilados ordenados y el tiempo óptimo:

```
def analizar_rivales(videos_rivales):
  videos_ordenados = ordenar_rivales(videos_rivales)
  tiempo_analisis = calcular_tiempo_total_de_analisis(videos_ordenados)
  return tiempo_analisis, videos_ordenados
```

2.2. Optimalidad del algoritmo

Expondremos un breve ejemplo para explicar por qué la solución a la que llegamos es óptima.

Los datos que usaremos son:

s_i	a_i
5	1
3	3
1	8

Tomamos este set de pruebas y graficamos los intervalos de tiempo tanto de Scaloni como de los ayudantes para visualizar los distintos escenarios.

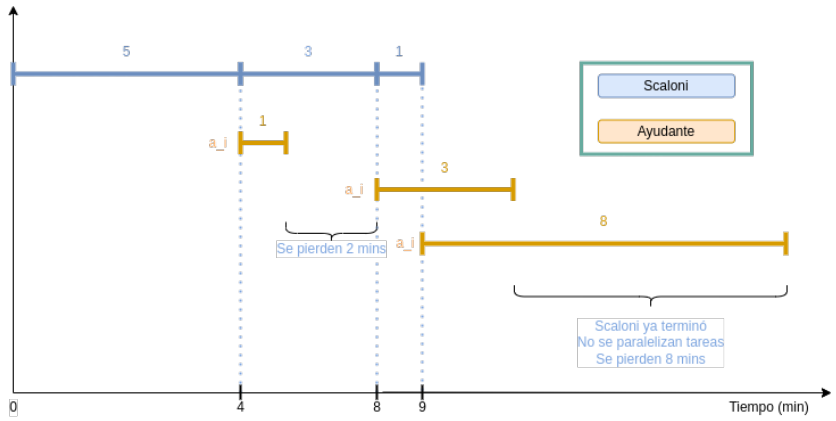

Figura 1: Videos sin un orden específico

Observamos que se pierde mucho tiempo después de que Scaloni finaliza su análisis.

Este tiempo podría ahorrarse si la tarea del ayudante que más tarda estuviese al principio, permitiendo así una mayor paralelización del trabajo. Mientras Scaloni continua analizando y delegando tareas, el ayudante con mayor tiempo de finalización podrá continuar su análisis en paralelo.

Movemos la tarea del ayudante que más tarda al principio y volvemos a analizar.

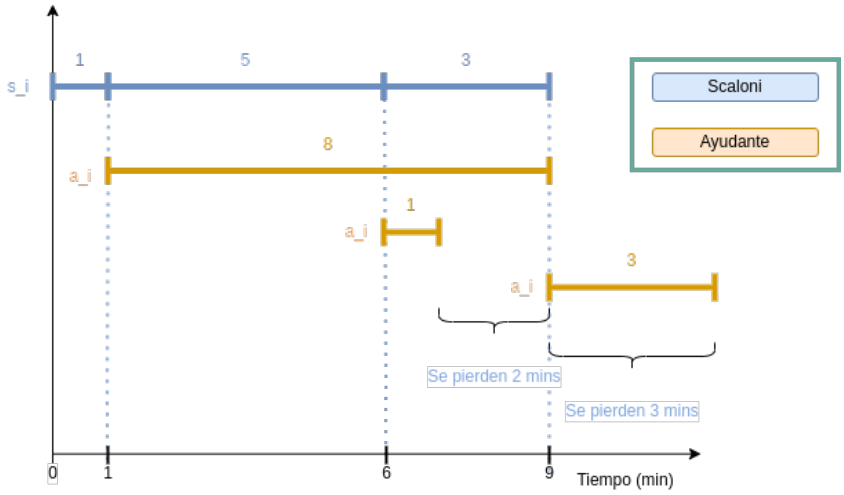

Figura 2: Ayudante con mayor tiempo al principio

Notamos que mejoró substancialmente el tiempo de análisis total, pero es claro que puede mejorarse apuntando, no solo a paralelizar lo más posible las tareas, sino también a minimizar el tiempo excedente de los ayudantes respecto del total de Scaloni.

Para eso, intercambiamos el orden de los últimos dos compilados.

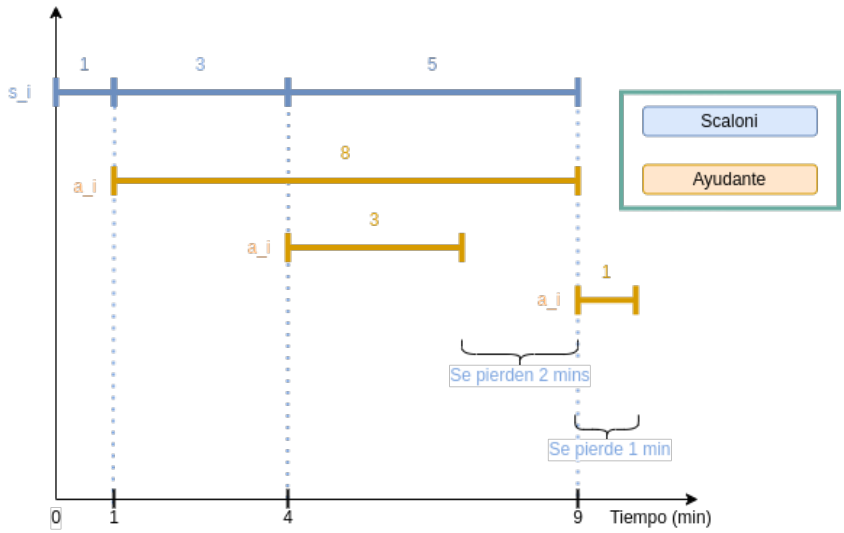

Figura 3: Videos ordenados por ayudante, de mayor a menor

Por último, vemos que los videos quedaron ordenados de mayor a menor según a_i y que el tiempo total es el óptimo.

3. Complejidad

Para calcular la complejidad del algoritmo propuesto podemos analizarlo en dos partes:

1. El ordenamiento de los compilados según el tiempo que tarda cada ayudante en estudiarlos.
2. El cálculo del tiempo óptimo a partir del arreglo ordenado.

Según la documentación de `Python`, la función `sorted()`[1] utiliza el algoritmo de ordenamiento **Timsort** que tiene una complejidad de: $\mathcal{O}(n \log n)$

Luego, para encontrar el tiempo mínimo en el que pueden analizar los compilados, recorremos el arreglo de n elementos una vez, realizando comparaciones numéricas en $\mathcal{O}(1)$. Siendo la complejidad de esta sección: $\mathcal{O}(n)$.

Por lo tanto, la complejidad total de nuestro algoritmo Greedy es de:

$$\mathcal{O}(n \log n)$$

[1] Python HOWTOs

4. Mediciones

Para programar nuestra solución elegimos el lenguaje `Python`.

Generamos set de datos con valores pseudo aleatorios usando el módulo `random` que exportamos en archivos de texto con el módulo `csv`.

Para las pruebas realizadas, medimos el tiempo de ejecución treinta veces para cada set de datos usando el módulo `time`. Estos valores se promediaron y el resultado lo usamos para realizar los gráficos usando la biblioteca `Seaborn`.

4.1. Tiempo de ejecución vs cantidad de rivales

Para el estudio del tiempo de ejecución en función de la cantidad de rivales (n), generamos set de datos con valores enteros pseudoaleatorios para a_i y para s_i de hasta diez mil elementos con intervalos de cien elementos.

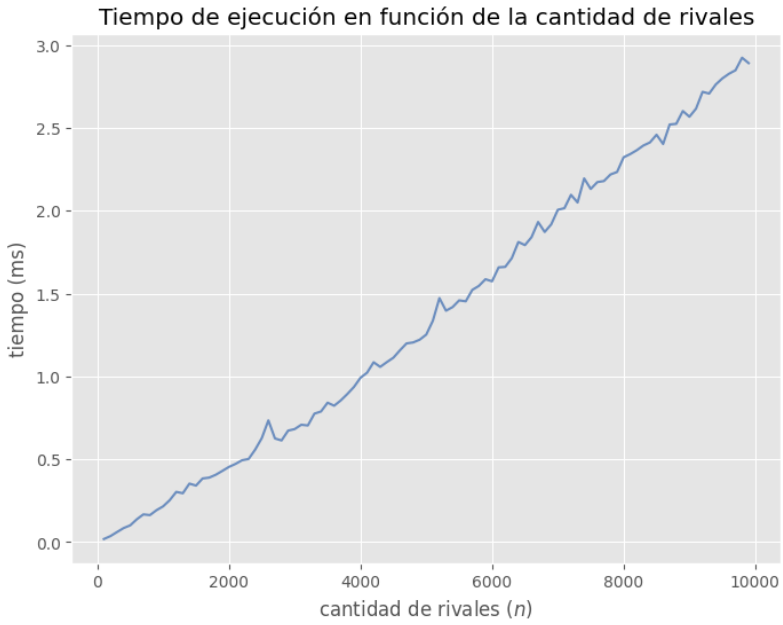

Figura 4: Tiempo de ejecución según cantidad de rivales

Se puede apreciar que el algoritmo tiene una tendencia $nlog(n)$ en función del tamaño de la entrada, lo que coincide con la complejidad teórica calculada anteriormente.

4.2. Variabilidad

En esta sección analizaremos brevemente el efecto de la variabilidad de s_i y los tiempos de ejecución del algoritmos. Generamos set de datos con valores pseudo aleatorios usando el módulo `random` acotando a un rango fijo los valores de a_i y variando en tres rangos s_i: 1-100, 1-100000, 1-100000000.

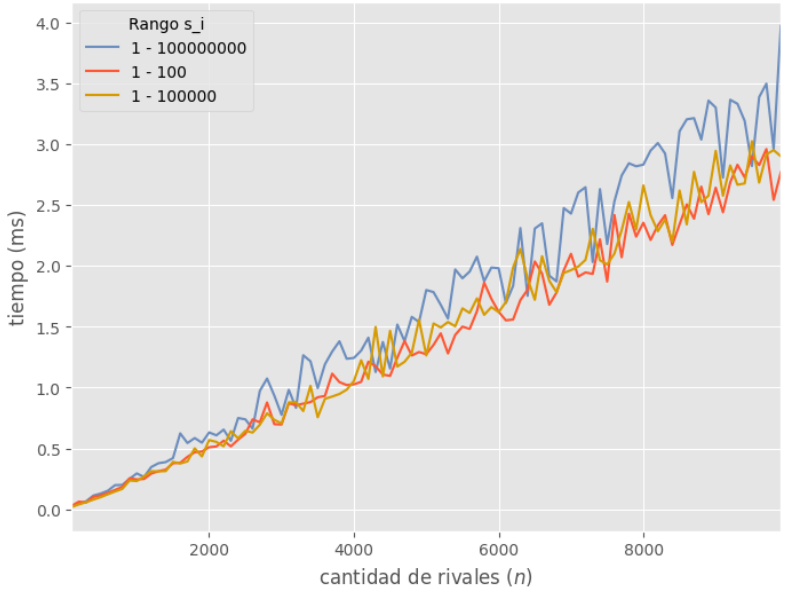

Figura 5: Tiempo de ejecución para diferentes rangos de variabilidad de s_i

Teniendo en cuenta el gráfico de la sección 4.1, podemos observar que, al variar los tiempos de Scaloni entre rangos de 1 a 100000000, no hay cambios significativos en los tiempos de ejecución.

Análogamente, generamos los sets de datos para a_i y visualizamos su variabilidad y los tiempos de ejecución:

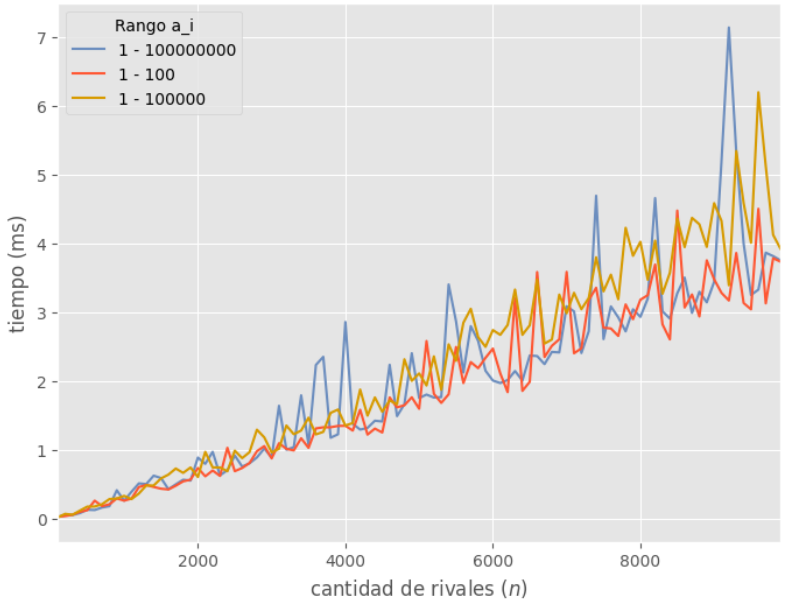

Figura 6: Tiempo de ejecución para diferentes rangos de variabilidad de a_i

5. Conclusiones

En este trabajo práctico, pudimos aplicar la técnica de diseño de algoritmos *Greedy* para resolver un problema de *scheduling*, con el objetivo de minimizar el tiempo que le llevaría a Scaloni y a su equipo de trabajo analizar los próximos rivales de la Selección Argentina.

Luego de realizar nuestro análisis y pruebas, concluimos que el algoritmo diseñado logra la solución óptima ordenando los a_i de mayor a menor, descartando que el orden de los s_i afecten el calculo del tiempo.

Finalmente, pudimos comprobar que la solución propuesta es óptima y que la complejidad del algoritmo es $O(nlogn)$.