Universidad del Valle

Facultad de Ingeniería

Escuela de Ingeniería de Sistemas y Computación

Estrategias de Optimización en la Distribución de Copias de Libros entre Escritores: Enfoques Voraz y Dinámico para Minimizar Tiempos de Copiado

Carlos Maricio Tovar Parra - 1741699

Profesor

Jesús Alexander Aranda Bueno Ph.D.

Curso

Fundamentos de Análisis y Diseño de Algoritmos (750094M)

19 de diciembre de 2023

Índice

1. Introducción

En el contexto de la reproducción manual de libros, previo a la aparición de las fotocopiadoras, nos enfrentamos al desafío de mejorar la distribución eficiente de la tarea entre varios escritores. Esta necesidad surge al intentar replicar una obra teatral dividida en múltiples libros para su presentación en un festival. En este informe, exploramos dos estrategias algorítmicas, voraz y dinámica, con el objetivo de minimizar el tiempo total empleado en la copia. A través de un enfoque práctico, analizamos las complejidades algorítmicas y comparamos el rendimiento de ambas estrategias en diversas pruebas. Dado que los detalles y la especificación del problema ya se encuentran en el enunciado del proyecto, este informe se centra exclusivamente en la presentación de las soluciones y su análisis, para las cuales se usará el lenguaje de programación Python.

2. La entrada

2.1. Formato de la Entrada

Las entradas válidas para los algoritmos que se van a plantear son archivos de texto .txt con la siguiente estructura:

```
n m
nombrelibro1 paginas1
...
nombrelibrom paginasm
```

donde n es el número de escritores y m el número de libros. La siguientes m lineas del archivo están compuestas, cada una del m-esimo volumen de la obra seguido de su cantidad de páginas.

2.2. Lectura de la Entrada: $O(m)$

La siguiente función en Python, llamada `leer_archivo`, se encarga de procesar un archivo de entrada que contiene información necesaria en el formato anteriormente mencionado para dar solución al problema

```
1  def leer_archivo(archivo):
2      with open(archivo, 'r') as file:
3          lines = file.readlines()
4          n, m = map(int, lines[0].split())
5          libros = ['name'] * m
6          paginas = [0] * m
7          for i in range(m):
8              libros[i], paginas[i] = lines[i+1].split()[0],
9              int(lines[i+1].split()[1])
10     return n, m, libros, paginas
```

La asignación de m y n, siempre en la primera línea, es constante. A continuación, se inicializan dos vectores: uno para los nombres de los libros y otro para sus respectivas páginas. Estos vectores se llenan en un bucle `for` que itera hasta m, la cantidad de libros. Por lo tanto, la complejidad de este algoritmo es $O(m)$

3. Solución Voraz $O(m)$

3.1. Idea General

La estrategia principal del algoritmo voraz propuesto es asignar secuencialmente los libros a un escritor y en cada asignación comprobar si la suma acumulada de las páginas asignadas hasta el momento supera un límite de trabajo estimado. Este valor se obtiene al dividir la suma total de páginas entre el número de escritores disponibles. La finalidad de este enfoque es asegurar una distribución equitativa de la carga de trabajo, considerando especialmente la minimización del tiempo dedicado del autor con mayor número de páginas asignadas. Este valor se utiliza como una guía para los intervalos de asignación de libros entre los escritores, y funciona porque proporciona una estimación inicial que tiende a equilibrar la carga de trabajo de manera efectiva.

Por ejemplo, para la siguiente entrada

```
3 4
Libro1 400
Libro2 200
Libro3 200
Libro4 300
```

El límite de trabajo estimado sería $(400 + 200 + 200 + 300)/3 = 900/3 = 300$.

Como ya se mencionó, después de hacer cada asignación secuencial se comprueba si las hojas asignadas al escritor actual han sobrepasado el límite de trabajo estimado y se busca tomar la mejor decisión en ese punto: Se evalúa si iniciar una nueva asignación para el siguiente escritor sería más beneficioso que continuar la asignación actual o no. La lógica detrás de esta decisión es la siguiente:

Se compara la diferencia entre el límite de trabajo estimado y la suma acumulada actual con la diferencia entre el límite de trabajo estimado y la suma acumulada más la cantidad de páginas del libro actual. Si la primera diferencia es menor o igual que la segunda, se decide continuar con la asignación actual al escritor. Si la segunda diferencia es menor que la primera, se decide iniciar una nueva asignación para el siguiente escritor.

En otras palabras, el objetivo es tomar la decisión que permita asignar libros al escritor de manera que la suma total de sus libros asignados sea lo más cercana posible al límite de trabajo estimado. Durante las asignaciones secuenciales se construye una lista de índices, en la cual se almacenan todos los puntos de transición entre las asignaciones de libros para cada escritor. Cada índice en esta lista indica el final de una secuencia de libros asignados a un escritor y el comienzo de la asignación siguiente. Esta lista será valiosa para la construcción de la solución.

3.1.1. El Algoritmo Voraz $O(m)$

```
1  def solucion_voraz(n, m, paginas):
2      if n == 0:
3          return float('inf'), [m-1]
4      if m == 0:
5          return 0, [0]
6      total_paginas = 0
7      for i in range(m):
8          total_paginas += paginas[i]
9      limite_estimado = total_paginas / n
10     indices = []
```

```
11      tiempo = paginas[0]
12      tiempo_total = 0
13      for j in range(1, m):
14          if tiempo + paginas[j] < limite_estimado:
15              tiempo += paginas[j]
16              indices.append(j)
17          elif limite_estimado - tiempo <= tiempo + paginas[j] - limite_estimado:
18              if tiempo_total < tiempo:
19                  tiempo_total = tiempo
20              tiempo = paginas[j]
21              indices.append(j-1)
22          else:
23              tiempo += paginas[j]
24              if tiempo_total < tiempo:
25                  tiempo_total = tiempo
26              tiempo = 0
27              indices.append(j)
28      return tiempo_total, indices
```

En primer lugar se hacen las comprobaciones en el caso de que no hayan libros o no hayan escritores y si alguna de estas es cierta, el algoritmo termina debido a que en ninguno de esos dos casos se pueden hacer asignaciones, esto es constante.

En caso de que no entre a ninguno de estos casos, se continúa con la ejecución del algoritmo: El primer ciclo `for` se encarga de sumar las páginas de todos los libros para a continuación encontrar el límite estimado, esto es $O(m+1) = O(m)$. Las siguientes líneas son inicializaciones para el vector de índices, la variable del tiempo, que sirve para almacenar el trabajo del escritor al que se le está asignando libros en cada punto y la variable del tiempo total que sería parte de la solución del problema, esto es constante.

El segundo ciclo `for` que recorre los todos libros y guarda los índices de transición según las decisiones voraces descritas en 3.1, tiene una complejidad de $O(m)$, ya que cada libro se examina y asigna una única vez. Por lo cual, el algoritmo `solucion_voraz` es $O(m)$

3.2. Escribiendo la Solución $O(min(n, m))$

```
1   def escribir_solucion(n, m, solucion, libros, filename):
2       out_filename = filename.split('.')[0] + '_out.txt'
3       with open(out_filename, 'w') as file:
4           file.write('Tiempo total: {}\n'.format(solucion[0]))
5           if n == 0:
6               file.write('No hay escritores!!!')
7           elif m == 0:
8               file.write('No hay libros!!!')
9           elif n < m:
10              for i in range(m):
11                  file.write('Escritor {}:'.format(i+1))
12                  file.write(' '.join(libros[i:i+1]) + '\n')
13              file.write('Quedan {} escritores sin libros asignados'.format(m-n))
14          else:
```

```
15                for i in range(n):
16                    file.write('Escritor {}:'.format(i+1))
17                    if i == 0:
18                        file.write(' '.join(libros[0:solucion[1][0]+1]))
19                    if i == n-1:
20                        file.write(' '.join(libros[solucion[1][i-1]+1:]))
21                    else:
22                        file.write(' '.join(libros[solucion[1][i-1]+
23                        1:solucion[1][i]+1]) + '\n')
```

Para la construcción de la solución, se empieza escribiendo en la primera línea del archivo el tiempo total devuelto por la función `solución_voraz`. Luego se contemplan los casos base en los que no hayan libros o escritores, para los cuales el algoritmo terminaría siendo $O(1)$.

Si no se entra a los casos base, se escriben las asignaciones de los libros a los escritores, para lo cual también existe un caso especial, en el cual hay más libros que escritores. En este caso se hace un ciclo `for` hasta m que asigna secuencialmente un libro a cada escritor hasta que se terminen los libros y al final quedarían m escritores con un libro asignado y $n - m$ escritores sin libro asignado. En este caso el algoritmo sería $O(m)$

En cualquier otro caso se escriben las asignaciones usando `solucion` que es el vector de índices de transición. Esto se realiza mediante un bucle `for` que itera hasta n, que representa la cantidad de escritores. En cada iteración, se asignan al escritor correspondiente los libros hasta el índice límite indicado por el vector `solucion`. En este caso el algoritmo sería $O(n)$

Entonces, en general, el algoritmo para escribir las soluciones termina siendo $O(min(n, m))$

3.3. Complejidad General $O(m)$

En resumen, las funciones `leer_archivo` y `solucion_voraz` tienen una complejidad de $O(m)$ y la construcción de la solución tiene una complejidad de $O(min(n, m))$, donde n es la cantidad de escritores y m es la cantidad de libros. Por lo que la complejidad general de la solución voraz es $O(m) + O(m) + O(min(n, m))$, lo que termina siendo $O(m)$

3.4. Pruebas

Se generaron 24 pruebas con n=10 y aumentando gradualmente el valor de m para conocer el comportamiento del algoritmo cuando va creciendo la entrada

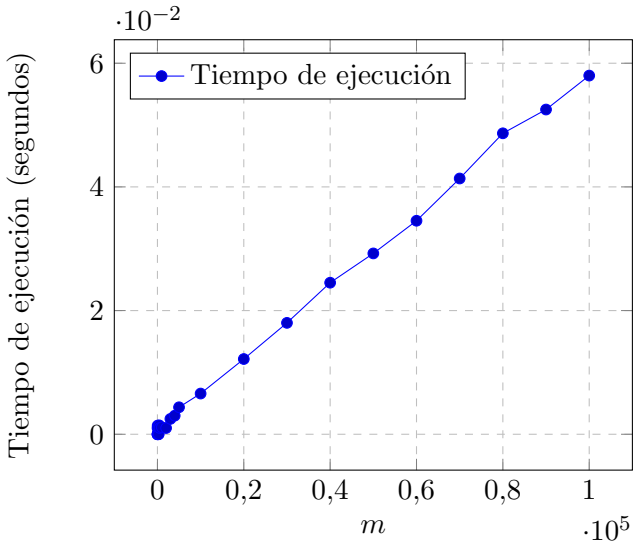

Figura 1: Gráfico de Tiempo de Ejecución del Algoritmo Voraz en función de m

Este gráfico de tiempos de ejecución confirma el análisis previamente hecho, que el algoritmo planteado usando una estrategia voraz tiene una complejidad algorítmica $O(m)$.

Para observar un poco mejor el comportamiento del algoritmo, se generó otro gráfico de tiempos de ejecución usando `matplotlib` a una mayor escala.

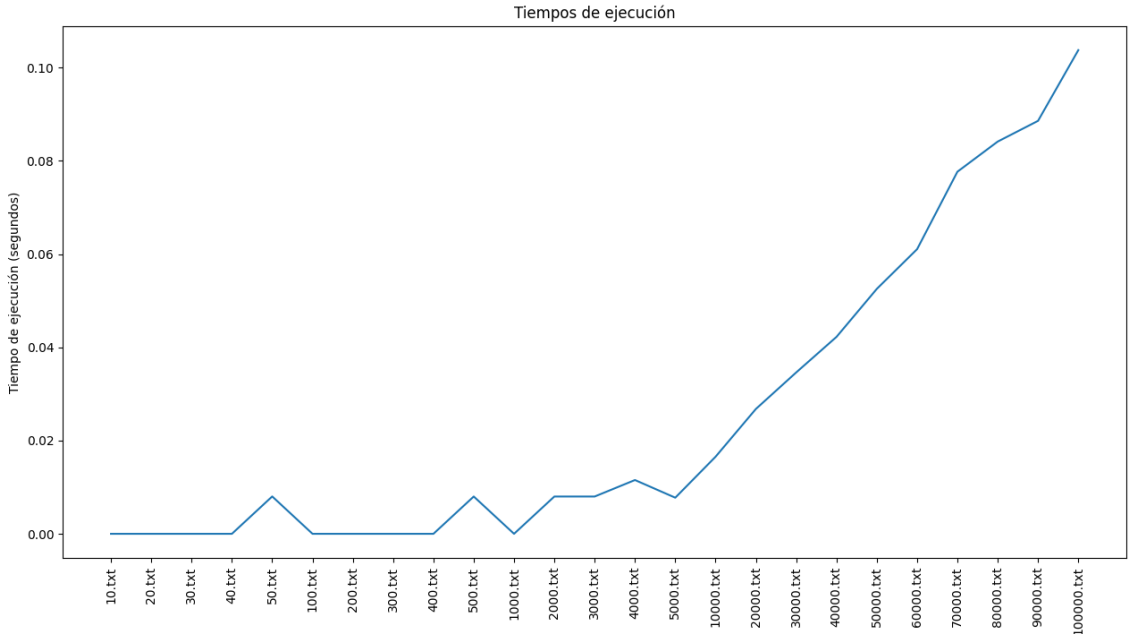

Figura 2: Gráfico de Tiempo de Ejecución del Algoritmo Voraz *matplotlib*.

Como se puede observar, el algoritmo tiene un comportamiento muy parecido a una función constante hasta aproximadamente $m = 10000$, y a partir de ese punto se comienza a evidenciar su comportamiento lineal. Lo que lo hace una solución altamente eficiente para entradas pequeñas e incluso medianas. Para entradas muy grandes sigue siendo una muy buena opción ya que es un algoritmo que crece proporcionalmente con la cantidad de entradas m. Afortunadamente, teniendo en cuenta el contexto del problema, hasta la fecha actual, y mucho menos hasta antes de la invención de las fotocopiadoras, no se ha registrado ninguna obra de teatro con más de 10000 volúmenes, por lo que esta es realmente una muy buena solución.

4. Solución Dinámica

4.1. Idea General

Inicialmente, se busca abordar el problema de manera recursiva evaluando todas las posibles asignaciones de libros a escritores y determinando el tiempo total mínimo necesario para realizar las copias. Esto es importante para entender la estructura del problema y lograr establecer una base para la optimización a través de la programación dinámica en la que se busca almacenar subproblemas resueltos previamente, lo que contribuirá a mejorar la eficiencia y evitar redundancias en el cálculo de soluciones.

4.2. Solución Exclusivamente Recursiva

```
1  def solucion_recursiva(n, m, paginas):
2      if m == 0:
3          return 0
4      elif n == 1:
5          return sum(paginas)
6      else:
7          min_tiempo = float('inf')
8          for i in range(1, m + 1):
9              tiempo_actual = max(sum(paginas[:i]),
10             solucion_recursiva(n - 1, m - i, paginas[i:]))
11             min_tiempo = min(min_tiempo, tiempo_actual)
12         return min_tiempo
```

El algoritmo `solucion_recursiva` aborda el problema de asignar libros a escritores de manera recursiva, evaluando todas las posibles asignaciones y determinando el tiempo total mínimo necesario para realizar las copias. Los casos base son cuando no hay libros por asignar o hay un único escritor al cual asignarle libros.

Cuando hay más de un libro por asignar y al menos un escritor disponible, la función entra en un bucle que itera sobre todas las posibles asignaciones de libros a escritores. Para cada asignación (i representa el número de libros asignados a un escritor), se calcula el tiempo total actual teniendo en cuenta el máximo entre las páginas del libro actual y el tiempo total calculado recursivamente para las asignaciones restantes. Se mantiene un registro del tiempo total mínimo encontrado hasta el momento `min_tiempo`. En cada iteración, se compara el tiempo actual con el mínimo actual, y se actualiza si el tiempo actual es menor. Al final de la función, se devuelve el tiempo total mínimo encontrado que es la cantidad de días total que se tardarán los escritores copiando los libros. La solución exclusivamente recursiva tiene la ventaja de ser fácil de entender e implementar,

sin embargo, puede volverse ineficiente para instancias grandes del problema debido al cálculo redundante de subproblemas inherente de las funciones recursivas. Además, como está planteada actualmente, no permite recuperar las mejores soluciones parciales para escribir las asignaciones en la solución final.

4.3. El Algoritmo Dinámico

```
1  def solucion_dinamica(n, m, paginas):
2      if dp[n][m] != -1:
3          return dp[n][m]
4      if m == 0:
5          return 0
6      elif n == 1:
7          return sum(paginas)
8      else:
9          min_tiempo = float('inf')
10         for i in range(1, m + 1):
11             tiempo_actual = max(sum(paginas[:i]),
12             solucion_dinamica(n - 1, m - i, paginas[i:]))
13             min_tiempo = min(min_tiempo, tiempo_actual)
14             dp[n][m] = min_tiempo
15         return min_tiempo
```

La solución dinámica propuesta se basa esencialmente en la solución recursiva anteriormente planteada, con la diferencia fundamental de que incorpora la técnica de memoización para evitar el recálculo de subproblemas. En cada nivel de recurrencia, las soluciones parciales se almacenan en una matriz dp de dimensiones nm, permitiendo que estas soluciones puedan ser consultadas en niveles de recurrencia posteriores para obtener resultados ya calculados.

Como se puede observar en la función, justo al iniciar se consulta si el cálculo que se requiere ya está almacenado en la matriz dp, que, vale la pena aclarar, se inicializa como una matriz global llena de -1.

En caso de que se requiera hacer el cálculo, se realiza siguiendo la misma lógica de la solución exclusivamente recursiva, pero se almacena en la matriz dp al finalizar, por si se llega a requerir en otro nivel de la recurrencia. La idea de esta matriz, además de optimizar la función, es permitir recuperar la solución a partir de sus soluciones parciales ya implementadas.

Sin embargo, a pesar de que esta solución dinámica propuesta funciona correctamente para determinar el tiempo total mínimo necesario, se ha identificado un desafío en la construcción de la matriz dp que afecta la capacidad de recuperar las asignaciones específicas de libros a escritores. Actualmente, la matriz no se está actualizando correctamente en cada cálculo como es de esperarse, lo que hace que, por el momento, la solución dinámica aún no permita obtener la solución completa del problema y a la vez sea igual de costosa que la solución exclusivamente recursiva.

A pesar de los esfuerzos realizados en la implementación de la solución dinámica, es evidente que aún hay aspectos que requieren atención y mejora para lograr una solución completamente óptima y que realmente se considere como una solución dinámica. Para continuar con la implementación de la solución dinámica, es necesario corregir la asignación de valores a la matriz, garantizando una actualización adecuada para cada asignación de libros a escritores que optimice la función y permita recuperar la solución.

5. Instrucciones de Ejecución

Para la solución voraz se debe ejecutar el siguiente comando en la raíz del proyecto, el cual se encarga de ejecutar la solución con las entradas `10.txt` hasta `100000.txt`, genera un archivo de salida para cada una de las entradas en la misma carpeta raíz, imprime en consola cada uno de los tiempos de ejecución y genera una gráfica con esos datos.

```
$ python voraz.py
```

Para la solución exclusivamente recursiva y la solución dinámica se debe ejecutar el siguiente comando, que va a ejecutar el algoritmo para la entrada dada. Ninguna de estas soluciones reconstruye la solución completa por las razones que ya se explicaron en 4. Por lo que únicamente se imprime en consola el tiempo mínimo de la solución y el tiempo de ejecución correspondiente. Estas funciones son bastante costosas y están poco optimizadas por ahora, por lo que para entradas grandes podría tardar varios minutos.

Solución exclusivamente recursiva:

```
$ python recursiva.py [filename]
```

Solución dinámica:

```
$ python dinamica.py [filename]
```

donde **filename** es el nombre del archivo de la prueba omitiendo la extensión `.txt`

6. Conclusiones

- La solución voraz demostró ser altamente eficiente para instancias pequeñas y medianas del problema. Su complejidad algorítmica lineal $O(m)$ permite un rendimiento rápido y práctico. El análisis de tiempo de ejecución respalda esta eficiencia, mostrando un comportamiento casi constante hasta un número significativo de libros.
- Aunque la implementación exclusivamente recursiva proporciona una solución correcta, su eficiencia se ve comprometida debido a la recursividad y la falta de memoización.
- En la solución dinámica, a pesar de hacer un intento por incorporar la memoización para evitar el cálculo redundante de subproblemas, aún hay inconvenientes en la construcción de la matriz *dp* y en la capacidad de recuperar la solución completa del problema, puesto que sí da la solución correcta en cuanto al tiempo mínimo pero no se puede reconstruir la solución ni recuperar valores ya calculados anteriormente para optimizar el algoritmo.
- En principio, y con las implementaciones y análisis que se han podido hacer hasta el momento, la solución voraz parece una opción óptima para este problema particular, teniendo en cuenta que es lineal y casi constante para casos pequeños y medianos. Sin embargo es necesario abordar los problemas de implementación de la solución dinámica para poder hacer un contraste y comparación más significativa en la complejidad y tiempos de ejecución que permita elegir el mejor algoritmo para este problema particular.