Análisis de Complejidad Computacional del Algoritmo de Havel-Hakimi

Autor: Fernando Osuna Manzo (Con la ayuda para redaccion y entendimiento de Gemini)

Fecha: 16 de octubre de 2025

Asunto: Análisis de la implementación con ordenamiento iterativo (O(n2logn)) y optimización con montículos (O(mlogn)).

---

1. Introducción e Intuición del Algoritmo

El teorema de Havel-Hakimi es un método fundamental en la teoría de grafos que determina si una secuencia finita de enteros no negativos puede representar los grados de un grafo simple. El algoritmo funciona de manera recursiva (o iterativa) y se basa en un principio de reducción: si la secuencia es gráfica, entonces también lo es una secuencia derivada más pequeña.

La intuición detrás del proceso se puede ilustrar con la **"analogía de la red social"**:

Imaginemos que la secuencia de grados `S` es una lista de "deseos de conexión" de n personas en un evento. La persona p1 es la más sociable, con un deseo de conexión d1. El algoritmo postula que para satisfacer el deseo de p1, debemos conectarla con las siguientes d1 personas más sociables. Si esta conexión es posible, el problema se reduce a verificar si el resto de las conexiones deseadas (con los deseos ya actualizados) forman una red válida.

El paso crucial y computacionalmente costoso es que, después de cada "ronda de presentaciones", la "popularidad" (el grado) de los participantes cambia. Quien era el segundo más sociable puede ahora ser el quinto, o viceversa. Por lo tanto, para aplicar el mismo principio en la siguiente iteración, debemos **reevaluar y reordenar** la lista para encontrar al nuevo individuo más sociable. Esta necesidad de reordenamiento en cada paso es el núcleo de la complejidad de la implementación ingenua.

---

2. Análisis de Complejidad de la Implementación con Ordenamiento Iterativo

Analicemos el costo computacional de la implementación que utiliza un ordenamiento completo en cada iteración del bucle principal.

2.1. Costo de una Sola Iteración (k)

Supongamos que el algoritmo se encuentra en la iteración k y que la longitud de la secuencia de grados restante es $m=n-(k-1)$. Los pasos dentro del bucle son:

1. **Ordenamiento Descendente**: El paso más demandante es ordenar la secuencia de m grados. Utilizando un algoritmo de ordenamiento eficiente basado en

comparaciones (como Merge Sort o Heap Sort), el costo es:
$C_{sort} = O(m \log m) = O((n-k)\log(n-k))$

2. **Extracción del Grado Máximo**: Se extrae el primer elemento, d_1, de la lista ordenada. Esto es una operación de $O(1)$.
3. **Actualización de Grados**: Se resta 1 a los siguientes d_1 elementos de la lista. Esta operación requiere d_1 sustracciones.
   $C_{update} = O(d_1)$
   En el peor de los casos, d_1 puede ser del orden de m, por lo que el costo es $O(m) = O(n-k)$.

El costo total de la iteración k, C_k, está dominado por la operación de ordenamiento.

$$C_k = C_{sort} + C_{update} = O((n-k)\log(n-k)) + O(n-k) = O((n-k)\log(n-k))$$

2.2. Sumatoria de Costos Totales

El bucle principal se ejecuta, en el peor de los casos, $n-1$ veces. La complejidad total, $T(n)$, es la suma de los costos de cada iteración, desde $k=1$ hasta $n-1$.

$$T(n) = \sum_{k=1}^{n-1} C_k = \sum_{k=1}^{n-1} O((n-k)\log(n-k))$$

Para resolver esta sumatoria, podemos acotar su valor. El término más grande de la suma ocurre en la primera iteración ($k=1$), donde el costo es $O(n \log n)$. El término más pequeño ocurre en las últimas iteraciones. Podemos establecer un límite superior aproximando cada término por el costo de la primera iteración:

$$T(n) \leq \sum_{k=1}^{n-1} O(n \log n)$$

Dado que $O(n \log n)$ no depende de la variable de la sumatoria k, podemos tratarlo como una constante dentro de la suma:

$$T(n) \leq O(n \log n) \sum_{k=1}^{n-1} 1 = O(n \log n) \cdot (n-1)$$

Expandiendo el resultado, obtenemos la complejidad final:

$$T(n) = O(n^2 \log n)$$

2.3. Ejemplo Concreto (n=8)

Consideremos la secuencia inicial $S = \{7,6,5,4,3,2,1,0\}$.

- **Iteración 1 (n=8)**:
  - La lista ya está ordenada (costo de verificación/ordenamiento aprox. $8 \log 8 = 24$ ops).
  - Tomamos $d_1 = 7$. Restamos 1 a los siguientes 7 elementos.
  - Nueva secuencia (antes de eliminar el 7): $\{0,5,4,3,2,1,0,-1\}$. La presencia del -1 indica que la secuencia no es gráfica. Pero para el análisis, sigamos. La secuencia a reordenar es $\{5,4,3,2,1,0,-1\}$.
- **Iteración 2 (n=7)**:
  - Lista: $\{5,4,3,2,1,0\}$ (ignorando el negativo).

    - Costo de ordenar: aprox. $6\log 6 \approx 15.5$ ops.
    - Tomamos $d_1=5$. Restamos 1 a los siguientes 5 elementos.
    - Nueva secuencia: $\{3,2,1,0,-1\}$.
- **Iteración 3 (n=5)**:
    - Lista: $\{3,2,1,0\}$.
    - Costo de ordenar: aprox. $4\log 4=8$ ops.

Como se observa, el costo de ordenamiento se acumula en cada paso, validando el análisis de $O(n^2 \log n)$.

---

3. Optimización con un Montículo Máximo (Max-Heap)

El cuello de botella es evidente: reordenar toda la lista cuando solo necesitamos el elemento máximo y la capacidad de actualizar eficientemente otros elementos. Esta es una aplicación ideal para la estructura de datos **Max-Heap**.

Un Max-Heap mantiene el elemento más grande en la raíz y permite las siguientes operaciones de manera eficiente:

- `build_max_heap(list)`: Construir un heap desde una lista. **Costo: O(n)**.
- `extract_max()`: Extraer el elemento raíz. **Costo: O(logn)**.
- `insert(value)`: Insertar un nuevo elemento. **Costo: O(logn)**.

3.1. Algoritmo de Havel-Hakimi con Heap

1. **Inicialización**: Construir un Max-Heap con la secuencia de grados inicial.
    - Costo: O(n).
2. **Bucle Principal**: Mientras el heap no esté vacío: a. Extraer el grado máximo, d, del heap. * Costo: O(logk), donde k es el tamaño actual del heap. b. Si d es mayor que el número de elementos restantes en el heap, la secuencia no es gráfica. c. Extraer los siguientes d elementos del heap, guardarlos en una lista temporal, restarles 1 y reinsertarlos en el heap. * Cada `extract_max` cuesta O(logk). * Cada `insert` cuesta O(logk). * Costo total de esta sub-etapa: $d \times (O(\log k)+O(\log k))=O(d \log k)$.

3.2. Análisis de Complejidad de la Versión Optimizada

La complejidad total es la suma de la construcción inicial y el costo acumulado de las operaciones dentro del bucle. El número total de operaciones de "restar 1" a lo largo de toda la ejecución del algoritmo es igual a la suma de todos los grados, que por el *handshaking lemma* es 2m, donde m es el número de aristas del grafo resultante.

Cada una de estas 2m operaciones de decremento implica una actualización en el heap (una extracción e inserción), con un costo logarítmico.

- **Costo de Inicialización**: O(n)
- **Costo del Bucle**: $\sum d_i \times O(\log n)=2m \times O(\log n)=O(m \log n)$.

La complejidad total del algoritmo optimizado es:

$T(n,m)=O(n+m\log n)$

Para grafos dispersos, donde m es del orden de n ($m \approx n$), la complejidad es $O(n\log n)$, una mejora exponencial sobre la implementación ingenua. Para grafos densos, donde m es del orden de n^2 ($m \approx n^2$), la complejidad se acerca a $O(n^2\log n)$, pero en la práctica, el menor factor constante y la eficiencia de las operaciones del heap aún lo hacen superior.

---

4. Conclusión

El análisis demuestra que una implementación directa del algoritmo de Havel-Hakimi, que reordena la secuencia de grados en cada iteración, resulta en una complejidad de tiempo de $O(n^2\log n)$. Si bien es conceptualmente simple, es ineficiente para secuencias grandes. Al identificar el cuello de botella computacional (la necesidad repetida de encontrar y extraer el máximo), podemos emplear una estructura de datos adecuada, el Max-Heap, para optimizar el proceso. Esta optimización reduce la complejidad a $O(m\log n)$, lo que representa una mejora drástica y hace que el algoritmo sea viable para grafos mucho más grandes.