Punto 3 – Texto Mínimo Reconstruible

Identificación:

Juan Andrés Romero -202013449

Luccas Rojas -201923052

Explicación:

Para el problema del texto mínimo reconstruible se planeó en desarrollar un algoritmo greedy debido a la complejidad del problema. Se encontró que el ejercicio en cuestión puede reducirse a un problema tipo TSP (Travelling Salesman Person), donde cada subcadena es un vértice del grafo, los pesos entre nodos es el overlap o sobrelapamiento de letras entre ellas y se debe de encontrar un camino de costo máximo que pase por cada uno de los vértices una sola vez. Por lo tanto, se puede concluir que el texto mínimo reconstruible o *Shortest Superstring Problem* es un problema NP-Hard que no se puede resolver en tiempo polinómico (a menos de que P = NP, pero ese es otro debate). Con base en esto, se planeó el desarrollo de un algoritmo ávaro aproximado de decisión que no tenga una complejidad temporal tan elevada, pero que pueda llegar a dar una respuesta equivocada (que el String resultante no sea el mínimo). De esta manera se llegó a un algoritmo basado en 2 funciones auxiliares y una principal. Dentro de primeras, se encuentran overlap() e isBlank(), las cuales calculan el sobrelapamiento de dos strings y revisan si un String se encuentra vacío o no respectivamente. Overlap recibe como parámetro 2 strings cualquiera y devuelve tanto el valor del overlap (cuántas letras se sobrelapan como máximo entre los strings) y una cadena que es combinación de las entradas realizada por el overlap encontrado. En cuanto a la función principal, ésta utiliza 2 listas auxiliares, y 3 variables que representarán los resultados de cada iteración del ciclo principal. Aux, que representa la lista de strings que están siendo operadas y aux2, que representa una lista de superstrings cuyos String padre tienen el overlap máximo encontrado.

El pseudocódigo del algoritmo es el siguiente:

Copiar el arreglo de entrada en el primer arreglo auxiliar

Realizar un ciclo while mientras aux tenga una longitud diferente de 1

    Encontrar el par de strings en aux que mayor overlap tengan

    Reemplazar el par encontrado por la combinación de las strings

Una vez se tenga un solo String en aux, ese es el resultado.

Sin embargo, existen 2 casos en donde este procedimiento puede irse por un camino que no es. El primero ocurre cuando no existe ningún tipo de overlap entre las strings pasadas como parámetro, entonces el Superstring más pequeño es la concatenación de los substrings. Por otro lado, el segundo caso es cuando si existe un sobrelapamiento, pero las strings finales no tienen overlap entre ellas, esto quiere decir, que el algoritmo tomó el camino equivocado, y por tanto, es necesario evaluar otro. En este caso, se copia aux2 dentro de aux y se revisa si existe una String de la lista original que no haya sido fusionada en alguno de los strings de aux2. Luego se vuelve a realizar el ciclo anterior hasta dar con el resultado.

Análisis de complejidad:

El análisis de la complejidad temporal de este problema es mucho más difícil que el de los otros debido a que ésta depende mucho de la entrada y de las decisiones que toma el algoritmo. En principio, en la primera iteración, el algoritmo asigna variables (Complejidad constante) y llena una lista auxiliar con todos los elementos de la lista inicial O(n). Luego, revisa cuál es el overlap máximo entre todas las strings presentes con lo cual este proceso también es de orden O(n*n-1), donde n es el tamaño de la lista de palabras original. Sin embargo, revisar el overlap contiene un propio ciclo que recorre toda la substring buscando las letras que sean compartidas entre ambas cadenas introducidas en la función. Debido a esto, su propia complejidad en el peor caso es de O(k) donde k es el tamaño de las substrings, y adicionalmente ya que este proceso se realiza 2 veces en cada ejecución de la función (se revisan overlaps por izquierda y por derecha), la complejidad de overlap es de O(2k). Debido a esto, la primera comprobación de overlap máximo es de O(n(n-1)+ n(n-1)2k]). Una vez se encuentra esto, el algoritmo borra (O(n) en el peor caso) ambas subcadenas padre de la lista aux y agrega el resultado de juntarlas (operaciones constantes).

En este momento, la complejidad temporal total del algoritmo es O(n(n-1)+[n(n-1)2k]+ 2n]), sin embargo, posteriormente, se revisa si hubo overlaps en la ejecución, pero las strings restantes en aux no comparten overlap o si no hubo overlaps y tampoco se comparte overlaps entre las strings de aux. Aquí, en caso de que se cumpla la segunda, se recorre y concatena los elementos la lista de entradas, lo cual es O(n), pero en caso de que se cumpla la primera, se llena aux con los elementos de aux2 y se revisa si hay cadenas originales que no estén incluidas en alguna de las strings de aux2. Este proceso sería O(x+x*n) donde x es el tamaño de aux2, el cual nunca es igual o mayor a n (Cantidad de superstrings cuyos padres tenían el máximo overlap).

Ejemplo del segundo:

aux = [“baaa”, “aaab”]
entrada = [“aaa”,”aab”,”baa”,”bbb”]

La subcadena aaa se encuentra en ambas cadenas de aux, aab en la segunda, baa en la primera, bbb no se encuentra en ninguna, entonces se agrega a aux.

Con esto, el algoritmo se vuelve $O(n(n-1) + [\mathrm{n}(\mathrm{n}-1)2\mathrm{k}] + n)$ en el primer caso y $O(n(n-1) + [\mathrm{n}(\mathrm{n}-1)2\mathrm{k}] + x + xn)$ en el segundo. Debido a que la cantidad de veces que ocurre la primera opción es menor a la segunda (y la segunda tiene mayor complejidad), tomaremos esta última para la complejidad general. Finalmente se reinician las variables temporales y se vuelve ejecutar el ciclo, teniendo cada vez menos cadenas en aux hasta que solo quede una cadena, la cual es la respuesta del algoritmo. En promedio, este procedimiento se hace n-1 veces, pero se puede seguir extendiendo si el algoritmo toma caminos incorrectos (principalmente que si exista overlap, pero las strings finales a fusionar no tengan overlap entre ellas). En estos casos, se vuelve a operar todo.

En conclusión, en cuanto a complejidad temporal, el algoritmo desarrollado tiene una complejidad promedio de $n[n(n-1) + [\mathrm{n}(\mathrm{n}-1)2\mathrm{k}] + x + xn]$, pero puede llegar a extenderse hasta $2^n * [n(n-1) + [\mathrm{n}(\mathrm{n}-1)2\mathrm{k}] + x + xn]$.

Simplificando:

$$2^n * [(n^2 - n)(1 + 2k) + x + xn]$$

Por lo tanto el orden de complejidad superior del algoritmo es de: $O(2^n * [(n^2 - n)(1 + 2k) + x + xn]$

Lo cual se puede resumir en: $O(2^n * n^2)$

Por otro lado, en cuanto a complejidad espacial, el algoritmo se basa en un manejo de 1 lista principal, 2 listas auxiliares y variables de control. La lista principal es de tamaño n, aux es de máximo O(n) y aux 2 es menor a n, entonces sería O(n+n+n-1+C) donde C es una constante. Por lo tanto, la complejidad espacial del algoritmo es de O(3n+C) = O(n).

Comentarios:

Al investigar, encontramos que el problema es bastante, pero bastante difícil y es probado que es de tipo NP-Hard. Por esto, para no matarnos la cabeza ni el computador, decidimos intentar desarrollar un algoritmo greedy, que funciona en la mayoría de los casos y que tiene una complejidad aceptable. Además, nos dimos cuenta, de que en los pocos casos donde la String que devolvía no era la más pequeña, si se cambia el orden de entrada de las subcadenas, el algoritmo sí encontraba la subcadena más pequeña en esa iteración.