RESOLUCIÓN DE PROBLEMAS DE DISEÑO Y ANÁLISIS DE ALGORITMOS

Lázaro David Alba ,Max Bengochea ,José Carlos Pendas

Problema 1

En una futurística ciudad llena de robots, cada robot pertenece a una facción identificada por un color específico. Las piezas de los robots fueron creadas por varios fabricantes distintos, y algunos robots tienen piezas hechas por fabricantes en común. Los robots de diferentes facciones están buscando formar una alianza estratégica multicolor. Esta alianza debe ser un grupo de robots donde todos tengan, dos a dos, alguna pieza hecha por el mismo fabricante, y lo más importante: debe haber al menos un robot de cada color en esta alianza. Dado un grupo de robots, determine si es posible o no formar una alianza multicolor.

Modelación

En el problema planteado anteriormente, nos percatamos que se puede construir un grafo de una forma que explicaremos próximamente, de manera que cada instancia de este sea equivalente a las del problema del conjunto independiente. Por lo tanto, si logramos resolver el problema del conjunto independiente, también habremos resuelto nuestro problema. Esto nos indica que el problema planteado es reducible al problema del conjunto independiente.

Modelado del Problema

Para modelar este problema en un grafo, seguimos los siguientes pasos:

1. **Vértices (robots)**: Cada vértice del grafo representa un robot.
2. **Aristas (relaciones entre robots)**: Para todo par de robots u y v:
    - Si los robots u y v tienen el mismo color, trazamos una arista entre ellos.
    - Si los fabricantes de las piezas de los robots u y v son disjuntos (no comparten ningún fabricante), trazamos una arista entre ellos.

Características del Conjunto de Nodos

Dado este modelo, un conjunto de nodos en el grafo cumple con las siguientes características:

- Para todo par de nodos u y v, si no existe una arista entre ellos, significa que los robots tienen colores diferentes y comparten al menos un fabricante en común.

Objetivo

El objetivo es encontrar un conjunto de vértices que cumpla con las condiciones planteadas, es decir, que para todo par de robots del conjunto exista al menos un fabricante compartido y que los robots sean de colores distintos. Si encontramos un conjunto de tamaño k, donde k es el número de colores, entonces este conjunto representa una alianza multicolor.

Conjunto Independiente

Nuestro problema se reduce a encontrar un conjunto independiente en el grafo que cumpla con las condiciones anteriores.Un conjunto independiente es un conjunto de vértices en un grafo tal que ninguno de sus vértices es adyacente a otro. Es decir, es un conjunto V de vértices tal que para ningún par de ellos existe alguna arista que los conecte. En otras palabras, cada arista en el grafo contiene a lo más un vértice en V. El tamaño de un conjunto independiente es el número de vértices que contiene.

Un conjunto independiente maximal es un conjunto independiente tal que, al añadir cualquier otro vértice, este deja de ser independiente. Por otro lado, el conjunto independiente máximo corresponde al mayor conjunto independiente definible sobre un grafo dado. El problema de encontrar un conjunto con estas características se llama problema del máximo conjunto independiente y es NP-completo.

Observación

Observemos que un conjunto independiente de vértices equivale a un conjunto de robots que cumplen con las características de no ser del mismo color y tener un fabricante en común. Si este conjunto tiene un tamaño k, entonces representa una alianza multicolor; de lo contrario, no existe una alianza multicolor. Además, es fácil notar que el

conjunto independiente de tamaño máximo debe tener como máximo k vértices.

¿Por qué? Porque contamos con k colores. Si el conjunto contiene más de k vértices, necesariamente habrá vértices conectados entre sí, ya que compartirían el mismo color, por el principio del palomar.

Transformación de Conjunto Independiente a Cobertura de Vértices

Para abordar el problema de conjunto independiente, recurriremos al problema de *vertex cover* (cobertura de vértices), que es uno de los problemas más emblemáticos en la categoría de NP-completos. Una cobertura de vértices de un grafo es un conjunto de vértices que incluye al menos uno de los extremos de cada arista del grafo. Este problema fue uno de los 21 problemas NP-completos identificados por Richard Karp y, por lo tanto, es un clásico en la teoría de la complejidad computacional, como vimos en clases.

Relación entre Conjunto Independiente y Cobertura de Vértices

Una propiedad fundamental de estos problemas es que un conjunto de vértices forma una cobertura de vértices si y sólo si su complemento es un conjunto independiente.

Demostración

Supongamos que tenemos un conjunto de cobertura de vértices $C \subseteq V$, y su complemento, $V \setminus C$, no es un conjunto independiente. Esto implica que existen al menos dos vértices u y v en $V \setminus C$ tales que $(u, v) \in E$, es decir, comparten una arista en el grafo.

Si esta arista (u, v) conecta vértices en $V \setminus C$, entonces la arista no está cubierta por ningún vértice en C, lo que contradice la definición de cobertura de vértices. Formalmente, una cobertura de vértices $C \subseteq V$ de un grafo no dirigido $G = (V, E)$ es un subconjunto de vértices tal que para cada arista $(u, v) \in E$, al menos uno de sus extremos pertenece a C. Es decir:

$$(u, v) \in E \Rightarrow u \in C \vee v \in C$$

Supongamos que $V \setminus C$ es un conjunto independiente y que $C \subseteq V$ es su complemento, pero que C no es un conjunto de cobertura de vértices. Esto significa que existe al menos una arista $(u, v) \in E$ tal que ninguno de los dos vértices u ni v pertenece a C. En otras palabras, tanto u como v pertenecen a $V \setminus C$.

Sin embargo, esto contradice la suposición de que $V \setminus C$ es un conjunto independiente, ya que, por definición, un conjunto independiente no puede contener vértices que compartan una arista. Si existe una arista (u, v) con ambos vértices en $V \setminus C$, entonces $V \setminus C$ no es independiente.

Por lo tanto, si $V \setminus C$ es un conjunto independiente, C debe ser una cobertura de vértices, es decir, debe cubrir todas las aristas del grafo.

En consecuencia, el número de vértices de un grafo $G = (V, E)$ es igual a la suma del tamaño de su conjunto mínimo de cobertura de vértices y el tamaño de un conjunto independiente máximo.

Esta relación nos permite transformar el problema de encontrar un conjunto independiente máximo en el problema de buscar una cobertura de vértices mínima, y viceversa. Esta equivalencia es fundamental para el estudio y la resolución de problemas NP-completos en teoría de grafos.

Solución

La solución a este problema consiste en encontrar una cobertura de vértices mínima. A partir de esta cobertura, determinamos su complemento. Si el complemento tiene tamaño k, entonces existe una alianza multicolor; de lo contrario, no existe tal alianza.

La variante de decisión del problema de cobertura de vértices es NP-completa, lo que implica que, a medida que el tamaño del grafo crece, el tiempo de ejecución de cualquier algoritmo exacto aumenta exponencialmente. Esto lo hace inviable para grafos grandes.

Fuerza Bruta

Una forma exacta de resolver el problema es mediante **fuerza bruta**, generando todos los subconjuntos posibles de vértices y verificando si cubren todas las aristas. Si se encuentra un subconjunto mínimo que cubra todas las aristas, se tiene la solución exacta. Sin embargo, este método tiene una **complejidad exponencial**, con 2^V subconjuntos para un grafo con V vértices, por lo que solo es práctico para grafos pequeños.

Algoritmos Aproximados

Dado que la fuerza bruta no es viable para grafos grandes, recurrimos a **algoritmos aproximados** que proporcionan soluciones razonablemente buenas en tiempos más cortos. A continuación, presentamos un algoritmo aproximado para la cobertura de vértices:

Algoritmo Aproximado de Cobertura de Vértices

El problema de la cobertura de vértices óptima es una versión de optimización de un problema NP-completo. Sin embargo, encontrar una cobertura de vértices que sea casi óptima no resulta demasiado difícil."

Algoritmo: APPROX-VERTEX_COVER (G: Grafo)

- $c \leftarrow \{\}$
- $E' \leftarrow E[G]$
- Mientras E' no esté vacío, hacer:
  - Sea (u, v) una arista arbitraria de E'
  - $c \leftarrow c \cup \{u, v\}$
  - Eliminar de E' toda arista incidente en u o v
- Retornar c

Análisis

Es fácil ver que el tiempo de ejecución de este algoritmo es $O(V + E)$, utilizando una lista de adyacencia para representar E'.

Teorema

`APPROX-VERTEX-COVER` es un algoritmo 2-aproximado de tiempo polinómico, es decir, el algoritmo tiene una cota de razón de 2.

Objetivo

Dado que este es un problema de minimización, estamos interesados en el menor valor de c/c^* posible. Específicamente, queremos mostrar que:

$$\frac{c}{c^*} \leq 2 = p(n)$$

En otras palabras, queremos demostrar que el algoritmo `APPROX-VERTEX-COVER` devuelve una cobertura de vértices que es, como máximo, el doble del tamaño de una cobertura óptima.

Demostración

Sean los conjuntos c y c^* los conjuntos resultantes de `APPROX-VERTEX-COVER` y `OPTIMAL-VERTEX-COVER`, respectivamente. Además, sea A el conjunto de aristas seleccionadas en la línea 4.

Dado que hemos añadido ambos vértices, obtenemos:

$$|c| = 2|A|$$

Formalmente, como ningún par de aristas en A está cubierto por el mismo vértice de c^* (ya que, una vez que se selecciona una arista en la línea 4, todas las demás aristas incidentes en sus extremos se eliminan de E' en la línea 6), tenemos la siguiente cota inferior:

$$|c^*| \geq |A|$$

En la línea 4, seleccionamos ambos extremos, lo que da una cota superior en el tamaño de la cobertura de vértices:

$$|c| \leq 2|A|.$$

Dado que la cota superior es exacta en este caso, tenemos:

$$|c| = 2|A|.$$

Tomamos $|c|/2 = |A|$ y lo sustituimos en la ecuación anterior:

$$|c^*| \geq \frac{|c|}{2}.$$

Por lo tanto:

$$\frac{|c^*|}{|c|} \geq \frac{1}{2}.$$

Esto nos lleva a:

$$\frac{|c^*|}{|c|} \leq 2 = p(n),$$

lo que prueba el teorema.

Nota

La implementación en Python de este algoritmo está disponible en el repositorio asociado.

Problema 2

Ante la inminente visita de una figura importante, los habitantes de una ciudad entraron en pánico y decidieron reparar las calles por las que podría transitar. Sin embargo, no sabían con certeza cuál sería la ruta exacta, por lo que recurrieron a Max en busca de ayuda.

La ciudad está compuesta por n puntos importantes, conectados por calles cuya longitud es conocida. Se sabe que José, la figura importante, iniciará

su viaje en uno de estos puntos (s) y lo terminará en otro (t). Los ciudadanos desean saber cuántas calles (aristas) participan en **algún camino de distancia mínima** entre todos los posibles pares de puntos s y t.

Observación

Para resolver este problema, inicialmente se pensó en una solución basada en programación dinámica que calcule el costo mínimo entre cada par de vértices. La idea es mantener una matriz que registre la menor distancia encontrada entre cada par de puntos, junto con las aristas que participan en los caminos de que logran esa distancia mínima. Si se encuentra un nuevo camino con una menor distancia, se actualiza la solución reemplazando el conjunto anterior por el nuevo conjunto de aristas que minmiza la distancia. Si se halla otro camino con la distancia mínima, este se une al conjunto de aristas óptimos.

El primer enfoque consistió en modificar el algoritmo de **Floyd-Warshall**, que resuelve el problema de caminos mínimos en grafos con complejidad $O(V^3)$.

Idea principal

El algoritmo de Floyd-Warshall utiliza la programación dinámica para encontrar el camino más corto entre todos los pares de nodos de un grafo. La clave está en construir una solución de manera incremental, refinando las distancias mínimas paso a paso hasta alcanzar la solución óptima.

Conceptos clave

Matriz de distancias: El algoritmo mantiene una matriz dist[i][j], donde cada celda dist[i][j] representa la distancia mínima entre los nodos i y j. Inicialmente, la matriz se llena con los pesos de las aristas que conectan directamente los nodos. Si no existe una arista directa entre i y j, se inicializa con infinito (∞).

Relajación de caminos: Para cada nodo k, el algoritmo revisa si el camino entre i y j puede mejorarse pasando a través de k. Si el camino que pasa por k resulta más corto que el existente entre i y j, se actualiza la distancia en la matriz de distancias. La fórmula utilizada para esta actualización es:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$$

Esta operación se repite para todos los pares de nodos i y j, considerando cada nodo k como intermediario, hasta que todas las combinaciones posibles hayan sido evaluadas.

Otra idea que contemplamos fue la siguiente:

1. IDEA DEL ALGORITMO

Vamos recorriendo arbitrariamente todas las aristas y cada vez que encontramos un nodo, guardamos el camino para llegar a él desde todos los nodos que hemos visto. Si descubrimos una arista que conduce a un nodo ya descubierto, actualizamos el camino que tenemos con ese camino. En caso de actualización, actualizamos desde el nodo actualizado todos los caminos hacia el resto de nodos a través del nodo recientemente descubierto.

2. CORRECTITUD

Es un algoritmo que se basa en programación dinámica con la siguiente subestructura óptima:

Iniciamos un recorrido DFS desde el nodo 0 (aunque se puede comenzar desde cualquier nodo). Cada vez que descubrimos un nodo v que no ha sido visitado, le decimos a todos los nodos que hemos visitado lo siguiente: sea u el nodo desde el cual encontramos v, es decir, que acabamos de recorrer la arista (u, v), le decimos a cada nodo x que hemos visto, que la distancia de x a v es la distancia x hasta u + u hasta v más la distancia desde u hasta v. Garantizamos que la distancia hasta u está en nuestra subestructura óptima. Si v ya hubiera sido visitado, verificamos que la distancia desde cualquier x hasta v sea menor que la distancia desde x hasta u más la distancia desde u hasta v. En caso de que esta distancia se actualice, debemos actualizar cualquier camino desde x hacia cualquier otro nodo donde sea mejor llegar pasando por el nodo v recientemente actualizado, lo cual se garantiza en nuestra estructura, ya que guardamos todos los nodos que se han visitado con la mejor distancia hasta el momento y podemos obtener cualquier información de distancias en $O(1)$. Al terminar nuestro recorrido, habremos pasado por todas las aristas y en nuestra subestructura todos los caminos estarán actualizados.

3. COMPLEJIDAD TEMPORAL

Primero inicializamos nuestra estructura en un conjunto y una lista para las aristas descubiertas y

los nodos, respectivamente. Al recorrer cada arista y cada nodo tenemos un costo inicial de $O(V + E)$. Cada vez que pasamos por una arista, a través de la cual descubrimos un nodo nuevo, recorremos los nodos que tenemos para establecer ese camino hasta el nuevo nodo, lo cual hasta este punto nos aporta un costo de $O((V-1)^2)$, ya que existen $V-1$ aristas de modo que no se generen ciclos a partir de ellas, dado que en este punto solo analizamos los nodos que se descubren por primera vez.

En el caso de que lleguemos a un nodo ya visitado, preguntamos por actualizaciones de caminos, lo cual tiene un costo de $O(V)$, ya que pasamos por los nodos que tenemos y actualizamos en $O(1)$. En caso de actualizar el camino de u a v, a partir del v descubierto, actualizamos todos los caminos desde u hacia todos los nodos que tenemos pasando por v, lo cual aporta un costo de $O((V \times$ (todas las aristas que actualicen nuestro camino))), resultando en un costo final de $O((E-(V-1)) \times V) + O(V^2)$ en el peor caso. Nótese que en el caso de que nuestro grafo sea un árbol, la complejidad sería $O(V^2)$.

Comparación entre algoritmos

La principal diferencia entre el algoritmo de Floyd-Warshall y el algoritmo anterior es su eficiencia en grafos no densos. Floyd-Warshall tiene una complejidad fija de $O(V^3)$, independientemente de la cantidad de aristas en el grafo, lo que lo hace menos eficiente en grafos dispersos. En contraste, el algoritmo anterior, aplicado a un árbol, puede alcanzar una complejidad de $O(V^2)$, aprovechando mejor la estructura del grafo y resultando más eficiente en este tipo de casos. aunque en grafos densos nuesto algoritmo es menos eficiente, podemos mejorarlo usando distra dikstra y no dfs

Problema 3

Dada una cadena S de longitud N y un número entero K, se define lo siguiente:

Sea C el conjunto de todos los caracteres presentes en S. Una cadena S se denomina *buena* si, para cada sufijo de S, la diferencia entre las frecuencias de cualquier par de caracteres en C no excede K. En particular, si el conjunto C contiene un solo carácter, entonces la cadena es automáticamente considerada buena.

El objetivo es determinar si existe una reordenación de S que sea buena. Si existen múltiples reordenaciones válidas, se debe imprimir la reordenación lexicográficamente más pequeña. Si no existe tal reordenación, se debe imprimir -1.

Es importante recordar que un sufijo de una cadena se obtiene eliminando algunos (posiblemente ninguno) caracteres desde el inicio de la cadena. Por ejemplo, para la cadena $S = abca$, los sufijos son $\{a, ca, bca, abca\}$.

Observaciones

Desarrollamos una heurística greedy que se basa en construir la cadena de menor peso lexicográfico posible, asegurando que los sufijos resultantes sean balanceados. El enfoque radica en ordenar los caracteres de forma que cada subcadena desde la posición inicial hasta cualquier posición k ($k \leq N$) tenga el menor peso lexicográfico posible, cumpliendo la condición de balance de frecuencias para cada sufijo.

Para ello, es necesario identificar un criterio que permita colocar tantos caracteres como sea posible en la cadena, siempre en orden lexicográfico, de modo que cualquier símbolo colocado más a la izquierda tenga menor peso lexicográfico. No obstante, se debe garantizar que esto no rompa el balance de las frecuencias de los sufijos restantes.

La condición que debe cumplir una cadena es que, para cualquier sufijo de S, la diferencia entre la frecuencia máxima ($f_{\max}$) y la frecuencia mínima ($f_{\min}$) de los caracteres en C no exceda K, es decir, $f_{\max} - f_{\min} \leq K$. Si esta condición se cumple para la cadena completa, se sigue que cualquier prefijo de longitud $n-1$, formado al remover un carácter, también debe cumplirla.

Además, se observa que si esta condición debe cumplirse para $f_{\min}$, entonces también debe cumplirse para cualquier otra frecuencia f_i dentro de C. Por tanto, para cualquier carácter i, al intentar colocarlo en la cadena, verificamos que se tenga suficiente frecuencia de caracteres restantes para mantener el balance en el prefijo restante de la cadena.

La verificación consiste en garantizar que, al colocar un carácter, la nueva cadena resultante sigue cumpliendo $f_{\max} - (f_i - 1) \leq K$, lo que se puede reescribir como $f_{\max} - f_i < K$. Bajo esta idea, se colocan tantos caracteres como sea posible, siempre en el menor orden lexicográfico, de modo que el prefijo restante siga siendo balanceado.

Ejemplo

Consideremos la cadena $S = \texttt{aabbccc}$ y $K = 1$.

Inicialmente, calculamos las frecuencias de cada carácter:

$$(2, \texttt{a}), (2, \texttt{b}), (3, \texttt{c})$$

Ahora, procedemos con la construcción de la cadena:

- Empezamos por el carácter lexicográficamente menor, en este caso la letra `a`:

  $$f_{\max} - f_a = 1 \not< 1$$

  Por lo tanto, no podemos colocar una `a`. Lo mismo ocurre con `b`:

  $$f_{\max} - f_b = 1 \not< 1$$

  Tampoco podemos colocar una `b`. Sin embargo, para `c`:

  $$f_{\max} - f_c = 0 < 1$$

  Esto es verdadero, por lo que colocamos una `c`:

  $$\texttt{c}\_\_\_\_\_$$

  Actualizamos las frecuencias: $(2, \texttt{a}), (2, \texttt{b}), (2, \texttt{c})$.
- Repetimos el proceso. Ahora sí podemos colocar una `a`:

  $$f_{\max} - f_a = 0 < 1$$

  Por lo tanto, colocamos una `a`:

  $$\texttt{ca}\_\_\_\_\_$$

  Actualizamos las frecuencias: $(1, \texttt{a}), (2, \texttt{b}), (2, \texttt{c})$.

Observamos que, cuando la frecuencia de un carácter se reduce a 1, colocar dicho carácter no afecta el balance de los sufijos restantes, ya que ese carácter no aparecerá en los sufijos que se formen luego.

Bajo esta misma idea, seguimos colocando caracteres hasta completar la cadena, garantizando que en todo momento se mantiene el balance requerido para los sufijos.

Demostración de Correctitud

Demostraremos la correctitud del algoritmo por inducción, basándonos en los sufijos de la cadena.

Caso Base: Tamaño 1

Si la cadena tiene longitud 1, es buena por definición, ya que un sufijo de longitud 1 contiene un solo carácter. En este caso, la frecuencia de ese carácter es 1, por lo que la diferencia de frecuencias se calcula como:

$$f_{\max} - f_{\mathrm{i}} = 1 - 1 = 0 < K,$$

donde $K \geq 1$ y i pertenece al conjunto de letras C, que solo contiene un único símbolo. Notemos que si $K = 0$, esta condición se cumple únicamente para cadenas que no repiten símbolos. Por lo tanto, en este caso, la cadena es trivialmente buena.

Caso Base: Tamaño 2

Si la cadena tiene longitud 2, consideramos dos subcasos:

1. **Ambos caracteres son iguales:** Sea la cadena $S = s_1 s_1$. Los sufijos son $s_1 s_1$ y s_1. Para el sufijo $s_1 s_1$, tenemos:

   $$f_{\max} = 2, \quad f_{\min} = 2,$$

   lo que satisface la condición:

   $$f_{\max} - f_{\min} = 2 - 2 = 0 < K.$$

   Para el sufijo s_1, como solo hay un carácter, la diferencia de frecuencias también es 0. Así, en este caso, el algoritmo mantiene los caracteres en el orden que aparecen y, por lo tanto, la cadena es buena.
2. **Los caracteres son diferentes:** Sea la cadena $S = s_1 s_2$, con $s_1 \neq s_2$. Los sufijos son $s_1 s_2$ y s_2. Para el sufijo $s_1 s_2$:

   $$f_{\max} = 1, \quad f_{\min} = 1,$$

   así que:

   $$f_{\max} - f_{\mathrm{i}} = 1 - 1 = 0 < K,$$

   donde i es el menor elemento en orden lexicográfico, garantizado por el algoritmo. En el sufijo s_2, la diferencia de frecuencias también es 0. En este caso, el algoritmo coloca los caracteres en orden lexicográfico, garantizando que cualquier reordenación sea la más pequeña posible. Así, el algoritmo es correcto para cadenas de longitud 2.

Hipótesis de Inducción

Supongamos que el algoritmo funciona correctamente para cadenas de longitud n, es decir, que genera una reordenación buena de longitud n que cumple la condición:

$$f_{\max} - f_{\mathrm{i}} < K$$

para cada sufijo de tamaño n.

Paso Inductivo: Tamaño $n+1$

Ahora consideremos una cadena de longitud $n+1$. El algoritmo selecciona el caracter lexicográficamente menor disponible, garantizando que el balance permite que la siguiente sufijo de tamaño n se pueda crear, y lo coloca en la primera posición. Luego, se construye una cadena de tamaño n con los caracteres restantes. Por la hipótesis de inducción, sabemos que el algoritmo generará correctamente esta subcadena de tamaño n, asegurando que todos los sufijos de esta subcadena serán buenos.

Dado que el carácter inicial es el menor lexicográficamente, garantiza que cualquier sufijo que comience desde esta posición será óptimo en cuanto al orden lexicográfico. Además, la colocación de este primer carácter no afecta el balance de frecuencias de los sufijos de tamaño $n-1$, manteniendo siempre la diferencia:

$$f_{\max} - f_{\mathrm{i}} < K.$$

siempre sera possible ubicar un caracter puesto que el fmaximo - fmaximo es < k en el peor de los casos de que no exista ningun valor de i que cumpla el criterio el maximo lo cumple, podemos llegar a la conclusion de que si para la cadena completa se cumple la condicion anterior siempre se pude obtener una cadena valida

Conclusión

Por inducción, hemos demostrado que el algoritmo es correcto para cualquier longitud de cadena N. Esto se debe a que mantiene la condición de balance de frecuencias para los sufijos, garantizando que se puede generar una reordenación lexicográficamente más pequeña que cumple:

$$f_{\max} - f_{\mathrm{i}} < K$$

para todos los sufijos, y además se forma tomando los valores de menor valor léxico siempre que sea posible.

Análisis de Complejidad

El algoritmo utiliza dos estructuras principales: un *heap* máximo (montículo) y un diccionario (o tabla hash) que lleva el conteo de las frecuencias de los caracteres en la cadena.

Heap Máximo

Inicialmente, se construye un *heap* a partir de los pares $(s, \text{frecuencia})$, donde s es un carácter y frecuencia es la cantidad de veces que aparece en la cadena. Para cada actualización en la frecuencia de un carácter, el diccionario se actualiza de forma que siempre se garantiza que el elemento en la cima del *heap* (el máximo) tiene la mayor frecuencia y su instancia es única.

Cada operación de inserción o extracción en el *heap* toma $O(\log n)$, donde n es el número de caracteres únicos en la cadena. Como se realizan n actualizaciones de frecuencia, la complejidad de mantener el *heap* es $O(n \log n)$ en el peor caso.

Selección Lexicográfica

En cada iteración, el algoritmo selecciona el carácter de menor peso lexicográfico entre los caracteres disponibles. Este proceso se realiza n veces, una por cada posición en la cadena, y seleccionar el carácter más pequeño en términos lexicográficos tiene una complejidad de $O(|C|)$, donde $|C|$ es el tamaño del conjunto de caracteres posibles (el alfabeto).

Complejidad Total

El costo total del algoritmo es la suma de las siguientes operaciones:

- Mantener el *heap* máximo durante n actualizaciones, lo cual tiene una complejidad de $O(n \log n)$.
- Seleccionar el carácter de menor peso lexicográfico en cada una de las n iteraciones, lo que tiene una complejidad de $O(n|C|)$.

En el peor caso, cuando el tamaño del alfabeto es proporcional a n (es decir, $|C| = O(n)$), la complejidad total del algoritmo es:

$$O(n \log n) + O(n|C|) = O(n \log n) + O(n^2).$$

Por lo tanto, la complejidad en el peor caso del algoritmo es $O(n^2)$.