Roads 4 Coba

Juan Jose Muñoz, Ovidio Navarro Pazos

September 2024

1 Descripción del Problema

El país de Coba ha evolucionado, ahora tiene inicialmente N ciudades aisladas, donde la i-ésima ciudad tiene una significancia de A_i. El Anciano jefe de Coba quiere conectar todas las ciudades. Él puede construir una carretera bidireccional de longitud L ($L > 0$) desde la ciudad X a la ciudad Y si $(A_X \& A_Y \& L) = L$, donde & representa el operador AND bit a bit.

¿Cuál es la longitud total mínima de las carreteras que tiene que construir para conectar todas las ciudades en Coba? Imprime -1 si es imposible.

Nota:

Se dice que la ciudad X y la ciudad Y están conectadas si existe una secuencia de ciudades $C_1, C_2, \ldots, C_K$ ($K \geq 1$) tal que $C_1 = X$, $C_K = Y$, y existe una carretera desde C_i a C_{i+1} ($1 \leq i < K$). Todas las ciudades en Coba se dicen conectadas cuando cada par de ciudades en Coba está conectado.

1.1 Planteamiento de solución

Podemos modelar la ciudad de Coba como un grafo en el que cada vértice representa una ciudad, con su nivel de importancia, y las aristas representan los caminos entre ellas. Este sería un grafo ponderado, donde cada camino tiene una longitud asociada.

El problema que se nos plantea consiste en encontrar un árbol abarcador de costo mínimo, en el que todas las aristas cumplan con una condición específica. Para resolverlo, debemos obtener un subgrafo donde todas las aristas satisfagan la condición dada: $(A_x \& A_y \& L = L)$. Una vez obtenido este subgrafo con las aristas permitidas, procedemos a calcular el árbol abarcador de costo mínimo utilizando el algoritmo de Kruskal.

1.2 Correctitud del algoritmo

Vamos a demostrar que el algoritmo anterior permite dar una solucion correcta al problema planteado.

- Al hacer una iteracion por todas las aristas y seleccionar solamente las que cumplan la condición planteada se esta garantizando que el subgrafo seleccionado sea correcto para lo que se pide.

- Utilizando el algoritmo de Kruskar garanizamos que el árbol resultante sea de costo minimo

1.3 Por qué el algoritmo de Kruskar funciona?

El Algoritmo de Kruskal produce un árbol de expansión mínima de G.

Demostración: Consideremos cualquier arista $e = (v, w)$ agregada por el Algoritmo de Kruskal, y sea S el conjunto de todos los nodos a los que v tiene un camino en el momento justo antes de que se agregue e. Claramente $v \in S$, pero $w \notin S$, ya que agregar e no crea un ciclo. Además, no se ha encontrado aún ninguna arista de S a $V - S$, ya que cualquier tal arista podría haber sido agregada sin crear un ciclo, y por lo tanto habría sido añadida por el Algoritmo de Kruskal. Así, e es la arista más liviana con un extremo en S y el otro en $V - S$, y por lo tanto, pertenece a todo árbol de expansión mínima.

Entonces, si podemos mostrar que la salida (V, T) del Algoritmo de Kruskal es de hecho un árbol de expansión de G, habremos terminado. Claramente (V, T) no contiene ciclos, ya que el algoritmo está explícitamente diseñado para evitar la creación de ciclos. Además, si (V, T) no estuviera conectado, entonces existiría un subconjunto no vacío de nodos S (no igual a todo V) tal que no hay arista desde S a $V - S$. Pero esto contradice el comportamiento del algoritmo: sabemos que, dado que G es conexo, hay al menos una arista entre S y $V - S$, y el algoritmo añadirá la primera de estas que encuentre.

1.4 Complejidad del Algoritmo

Siguiendo los pasos descritos anteriormente, primero se calcula la complejidad de realizar una iteración por todas las aristas del grafo, verificando si se cumple la condición de las aristas. Esta operación, que simplemente recorre las aristas una por una, se completa en $O(E)$

Una vez completada esta etapa, procedemos a ejecutar el **Algoritmo de Kruskal**. El primer paso en este algoritmo es ordenar las aristas del grafo por su peso. Este proceso de ordenación tiene un coste computacional que depende del algoritmo de ordenación utilizado. Si se emplean algoritmos eficientes como *quicksort* o *heapsort*, el tiempo de ordenación será de $O(E \log E)$ En realidad, dado que $E \leq V^2$, donde V es el número de vértices, esta complejidad se puede expresar de manera equivalente como $O(E \log V)$, ya que $\log E$ es asintóticamente equivalente a $\log V$ en el peor de los casos.

El siguiente paso en el algoritmo de Kruskal es gestionar la conectividad de los nodos mientras se construye el Árbol de Expansión Mínima. Para ello, se utiliza **Union-Find** (o conjunto disjunto), que permite realizar dos operaciones clave: "unión" (merge) y "búsqueda" (find). Con una implementación eficiente de Union-Find que utilice *compresión de caminos* y *unión por rango*, cada operación de unión o búsqueda puede realizarse en un tiempo casi constante, de $O(\alpha(V))$, donde $\alpha(V)$ es la **función inversa de Ackermann**. Esta

función crece tan lentamente que, para todos los propósitos prácticos, se puede considerar constante incluso para tamaños grandes de grafo.

En conjunto, el tiempo total que toma el Algoritmo de Kruskal es dominado por el paso de ordenación de las aristas, que es de $O(E \log V)$. Las operaciones de Union-Find contribuyen con un término $O(E\alpha(V))$, que es mucho más pequeño y casi constante. Por lo tanto, la complejidad temporal total del Algoritmo de Kruskal es $O(E \log V)$.

En resumen, aunque el paso inicial de iterar sobre todas las aristas tiene una complejidad de $O(E)$, este no afecta el término dominante de la complejidad total, que sigue siendo $O(E \log V)$.